分类号 分类号 密级 保密级别

UDC 编号

中国科学院研究生院
硕士学位论文

PHG 中线性棱单元的实现及
三维电磁时谐场问题并行自适应数值模拟实验

崔涛

指导教师 张林波、研究员、博士

中国科学院数学与系统科学研究院

申请学位级别 硕士 学科专业名称 计算数学

论文提交日期 论文答辩日期

培养单位 中国科学院数学与系统科学研究院

学位授予单位 中国科学院研究生院

答辩委员会主席

摘　要

本文的主要内容是关于 Nédélec 线性四面体有限元在 PHG (Parallel Hierarchical Grid) 中的实现，以及关于时谐 Maxwell 方程组的并行自适应数值模拟实验。

PHG 是我们自主设计的一个基于网格二分细化、适合大规模分布式存储并行计算机的三维并行自适应软件平台。为了实现 Nédélec 线性四面体有限元，在本文的工作中，首先，我们改进了 PHG 中的部分模块使其支持向量基函数，并进行了大量的测试验证这些模块的正确性。其次，改进了 PHG 中数值积分函数的 cache 机制，该机制在保证程序效率的同时使得用户接口更加规范、简便，数值实验证明了我们的 cache 机制对有限元计算的有效性。最后，我们将棱单元实现细节及并行相关内容封装在 PHG 的库中。由于 PHG 代码本身的复杂性，这些工作具有相当的工作量和一定的难度，并且目前国际上尚没有公开发布类似软件，本工作中涉及的许多研究内容是探索性、创新性的。

在此基础上，利用 PETSc 提供的 LGMRES 求解器，我们在分布式存储并行计算机上利用 PHG 计算一批典型算例，包括三维电磁时谐 Maxwell 方程组自适应数值模拟等。PHG 提供一系列规范化的接口，对分布式网格管理、自由度管理、合成分布式线性系统、并行后验误差估计等一系列烦琐的代码进行了很好的封装，使得用户代码中不必关心与并行及网格操作相关的内容，做到与串行程序一样简洁。这些的数值算例验证了 PHG 所实现的自适应算法的有效性和代码的正确性，这些数值实验中包括 128 个结点、超过一亿个自由度的并行有限元计算。在已经进行的数值实验中，PHG 的各部分代码，包括本文所进行的扩展，表现出了很好的鲁棒性、计算效率和并行可扩展性。

关键词: 并行计算、自适应有限元、时谐场、线性棱单元、后验误差估计、四面体网格二分细化

Implementation of Linear Nédélec Element in PHG and Parallel Adaptive Simulation of 3-D Time-Harmonic Electro-Magnetic Problems

Cui Tao (majored in compuational mathematics)
Directed by Prof. Zhang Lin-bo

PHG (Parallel Hierarchical Grid) is a parallel adaptive finite element toolbox under development at the Academy of Mathematics and System sciences, Chinese Academy of Sciences, for parallel solution of partial differential equations (PDEs) using adaptive finite element method. This thesis presents the work on implementing the linear Nédélec element in PHG and numerical experiments with time-harmonic electro-magnetic problems.

PHG works with conforming tetrahedral meshes which are adaptively refined by bisectioning selected tetrahedra. It is designed for distributed memeory parallel computers. This work mainly consisted of tuning, testing and debugging modules and data structures of PHG to make them suitable for finite element computations using vector basis functions such as the linear Nédélec element. Some aspects of the internal design of PHG were also improved. For example, a more flexible cache mechanism for transparently caching values of basis functions and their gradients at the quadrature points was implemented, which simplifies the user interface for finite element applications, and the linear solver interface of PHG was enhanced for dealing with more complex problems. The linear Nédélec element was integrated into PHG and thoroughly tested. A set of new functions which are useful for solving the Maxwell's equations was implemented.

The implementation of linear Nédélec element was further justified by a number of numerical experiments with 3D time-harmonic electro-magnetic problems. Up to 128 cluster nodes were used to solve problems with up to 100 million degrees of freedom. For problems with strong singularities, expected convergence behaviour of the adaptive procedure based on a posteriori error estimates was observed. The computations show that our code is reliable, efficient, and has good parallel scalability.

Keywords: parallel computing, adaptive finite element, tetrahedral mesh, bisectioning refinement, Nédélec element, a posteriori error estimates

第一章　引言

偏微分方程可以用于模拟各种物理、化学和生物现象以及各种工程问题。但是这些方程很少能获得解析解，所以通常用数值近似的方法来获得这些方程的离散解。一种常用的求解偏微分方程的方法是在系统内用基于有限差分、有限体积或者有限元等方法的离散估计来代替偏微分，导出一个线性或非线性代数方程组，然后求解该代数方程组，最终得到方程的离散解。随着实际问题对计算精度的要求越来越高，人们对数值方法及计算能力也提出了更高的要求，电磁场数值分析便是其中之一。自适应有限元方法和并行计算方法以及并行计算机的发展，为挑战各种超大规模问题提供了可能。

§1.1　自适应有限元方法

在对很多物理和工程中的实际问题进行数值求解时，我们经常会遇到这样的问题：由于问题的局部奇异性而导致数值解的整体精度丧失。最简单的解决办法就是在整个计算区域均匀地增加更多的网格点，但是，这样会导致问题规模迅速扩大而超出计算机的存储和计算能力；一个更为可行的办法是在有奇性的区域增加网格点，同时保持非奇性区域网格的稀疏性，这样既可以达到求解问题所需要的精度，又可以节省计算资源。自适应方法使得解决这一问题成为可能。常用的自适应方法都是根据问题的奇性来调整网格或者局部基函数实现的：

- h 方法：通过网格的细化或粗化来实现自适应。
- p 方法：通过在局部增加或减少基函数来实现自适应。
- h–p 方法：结合上面两种方法来实现自适应。
- r 方法：移动网格方法，通过修改网格点的位置来实现自适应。

如果没有特别的说明，本文之后提到的自适应方法都指 h 方法。

Babŭska. I 和 Rheinboldt. W 在 1978 年提出了基于后验误差估计的自适应有限元方法 [2]。自适应有限元方法根据每个单元上的后验误差估计来判断奇性区域和需要在该区域增加的网格点数。Rüdiger Verfürth 在 [4] 中详细介绍了残量型的后验误差估计方法和相应的自适应算法。自适应有限元计算的基本过程如图 1.1 所示。

自适应方法的成功实现主要依靠几个因素：后验误差估计，细化和粗化的策略，自适应过程的停止准则以及各种参数的选取。后验误差估计是该方法的关

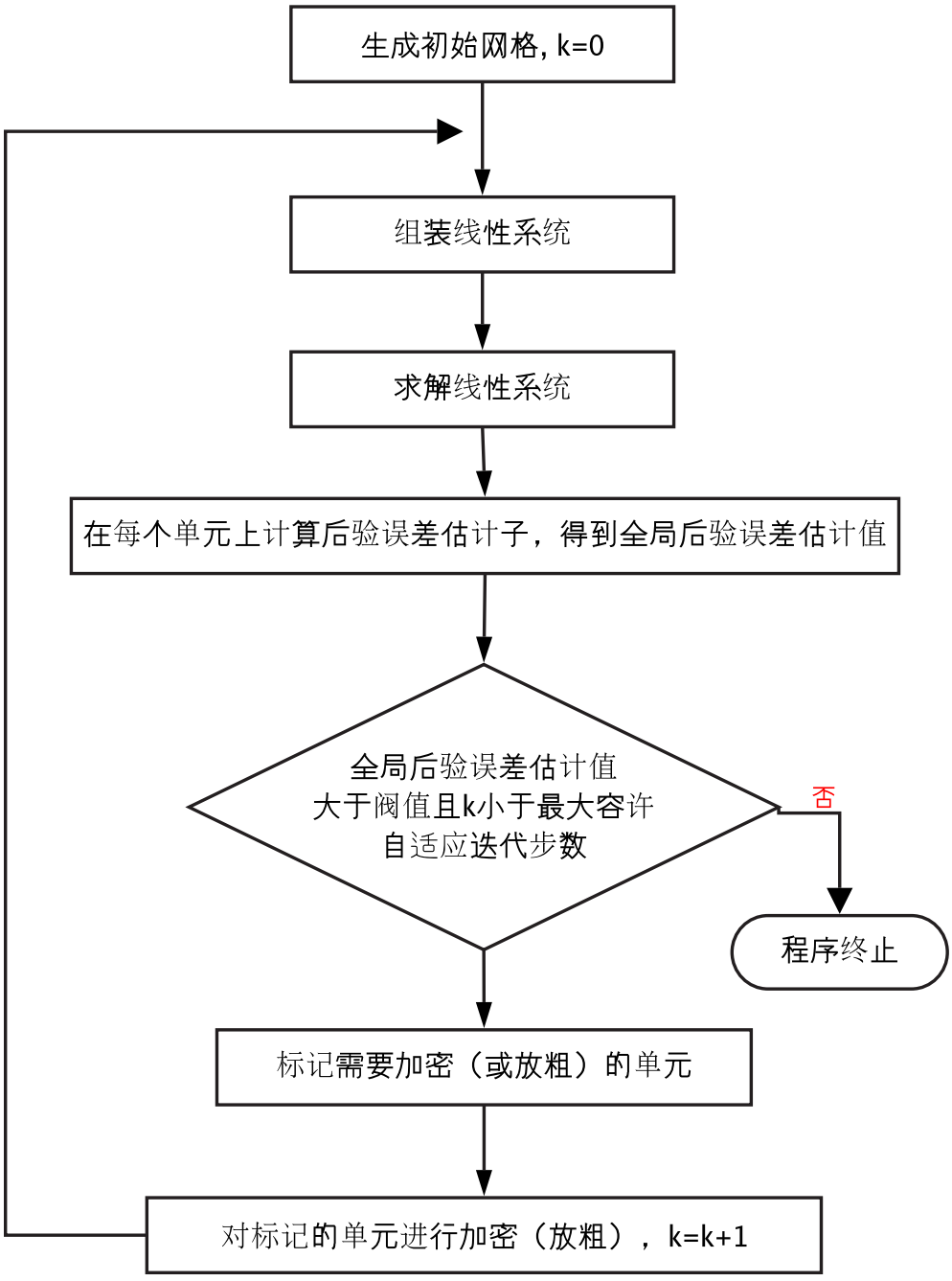

图 1.1 自适应有限元计算流程图

键，后验误差估计及其估计子的好坏直接影响到计算的效率。在得到了一个后验误差估计之后，自适应算法根据后验误差估计子来调整网格，使得在调整后的网格上的估计误差几乎平均分配，从而提高计算效率。

在给定后验误差估计子的前提下，如何选择单元来进行细化或者粗化将直接影响算法的收敛性和效率。一般来说，常用的细化策略有以下几种：假定 t 是协调剖分网格 $\mathcal{T}$ 中的任意一个单元，tol 是自适应终止的误差阀值，η_t 表示单元 t 上的后验误差估计子。

(1) **最大误差策略（Maximum strategy）：** 给定 $\gamma \in (0,1)$ 选取所有满足以下不等式的单元 $t \in \mathcal{T}$ 标记为细化

$$\eta_t > \gamma \max_{t \in Y} \eta_t$$

对于 2 范数的后验误差估计子一般选取 $\gamma = 0.5$

算法 1.1. 给定 $\gamma \in (0,1)$

```
procedure marking( T )
 η_max := max(η_t, t ∈ T)
forall t in T do
  if ( η_t > γ η_max )
    mark t for refinement
end for
end procedure
```

(2) **误差等分布策略（Equidistribution strategy）：** 设 N_t 为 $\mathcal{T}$ 中总的单元数，如果我们假设在所有单元上的误差平均分配，即 $\eta_t = \eta_{t'}, \forall t, t' \in \mathcal{T}$，那么

$$\eta = \left(\sum_{t \in \mathcal{T}} \eta_t^p \right)^{1/p} = N_t^{1/p} \eta_t \leq \text{tol}$$

$$\Longrightarrow \eta_t \leq \frac{\text{tol}}{N_t^{1/p}}$$

为了达到这种平均分配，我们选取满足以下不等式的单元，标记为细化。

$$\eta_t > \frac{tol}{N_t^{1/p}}$$

算法 1.2.

```
procedure marking( T )
forall t in T do
  if ( η_t > θtol/N_t^{1/p} ) then
    mark t for refinement
end for
end procedure
```

(3) **保证误差下降策略（Guaranteed error reduction strategy）：** 对于上面两种细化策略，尚未能从理论上证明其收敛性。Dörfler [21] 对于 Poission 问题给出了一种能保证相对误差减小的策略，基于如下假设：

- 问题的已知数据（例如：系数和右端项等）由当前网格上的值所决定，即可以忽略数值积分的误差。
- 被标记为细化的单元的每一条边至少要二分一次。

给定参数 $\theta \in (0,1)$，标记集合 $\mathcal{A} \subseteq \mathcal{T}$，使得 $\sum_{t\in\mathcal{A}} \eta_t^p \geq (1-\theta)^p \eta^p$

算法 1.3. 给定 $\theta \in (0,1); \nu \in (0,1)$

```
procedure marking( T )
η_max := max(η_t, t ∈ T)
sum := 0
γ := 1
while sum < (1-θ)^p η^p
  γ := γ - ν
  forall t ∈ T do
    if (t is not marked) then
      if ( η_t > γ η_max ) then
          mark t for refinement
        sum := sum + η_t^p
        end if
    end if
  end for
end while
```

```
end procedure
```

由于粗化网格时会产生额外的误差，而且粗化一个单元时会引起周围相邻单元的粗化。所以，只有当单元后验误差估计非常小的时候才可以标记该单元粗化，而且要保证粗化后产生的误差不会大于用来判定细化单元的误差阀值。对于定常问题，给定网格中所有单元的后验误差估计后，粗化策略一般有以下几种：

(1) **最大误差策略（Maximum strategy）：**给定参数 $\gamma > \gamma_c$，被粗化的单元满足:

$$\eta_t^p + \eta_{c,t}^p \leq \gamma_c \max_{t\in\mathcal{T}} \eta_t^p$$

(2) **误差等分布策略（Equidistribution strategy）：**给定参数 θ_c，被粗化的单元满足

$$\eta_t^p + \eta_{c,t}^p \leq \theta_c \frac{\mathrm{tol}}{N_t^{1/p}}$$

(3) **保证误差下降策略（Guaranteed error reduction strategy）：**给定 $\theta_c \in (0,1)$，粗化集合 $\mathcal{A} \in \mathcal{T}$ 中的单元，被粗化的单元满足:

$$\sum_{t\in\mathcal{A}} (\eta_t^p + \eta_{c,t}^p) \leq \theta_c^p \eta^p$$

本文中的例子如果没有特殊说明都将采用最大误差策略。

§1.2 并行数值计算软件

随着计算机硬件技术的快速发展，数值计算软件的应用已经深入到科学研究以及工业生产的各个领域。数值计算软件的发展也促进了理论分析的发展，它不仅成为验证理论分析结果的有效工具，而且能模拟出丰富具体的物理现象，揭示更多的问题，从而启发新的算法和新的思考。

但是随着问题规模的增长以及对计算结果精度要求的提高，串行程序越来越难以满足实际应用的需求。并行计算方法和并行计算环境的出现，为求解大规模科学和工程问题提供了一条途径。数值并行算法的研究逐渐成为很多科学研究和工程应用领域的核心问题，并行数值软件的研发也越来越受到更多人的关注。可以预见，并行自适应方法将成为今后很长一段时间内大家关注的焦点。基于并行自适应方法开发的数值计算软件包将会成为工程师以及科研人员手中的有力工具。

但是并行软件的开发比传统的串行程序更加复杂。对于同一个数值计算方法，串行算法的设计取决于该方法的数学本质，不会因为计算环境的不同而不同，串行程序的实现也大同小异。但是并行算法的设计却会因为计算环境的不同而有很大差别，并行程序的实现也相去甚远。这里需要考虑计算环境的类型和特点，如：是并行机还是向量机；是分布式存储类型还是共享存储类型；以及处理器的个数、网络环境等因素。

根据计算环境的不同，并行程序的设计模型大致可分为以下四类：

- 向量程序模型
  向量程序设计就是把标量程序中可以向量化的循环改成向量操作，常用的设计方式有两种：一、使用向量程序设计语言表示程序中可向量化计算的部分；二、利用编译系统的向量化功能将串行程序中可向量化计算的部分向量化。

- 数据并行模型
  利用数据之间的无关性，对数据作合理的安排，尽量使对数据的操作是局部的，不需要和其它结点进行数据交换。支持指令级的细粒度和语句、循环级的中粒度并行，适用于 SMP、DSM 和 MPP 并行机，最典型的编程语言是高性能 FORTRAN（HPF）。

- 共享存储模型
  所有的处理器拥有共同的物理内存，通过读 / 写公共存储器中的共享变量进行通信。支持线程级并行，适用于 SMP 和 DSM 并行机。目前最流行的共享存储程序设计标准是 OpenMP。

- 消息传递模型
  支持进程级的大粒度并行，具有很好的可扩展性，非常适合于大规模科学工程计算。各个处理器拥有独立的物理内存，互相之间不能直接访问，必须通过消息传递来交换数据。消息传递并行程序的设计要求用户很好地分解问题，合理组织数据的交换。当前，由于高速互联网络的出现，消息传递模型已经成为被广泛支持的并行程序设计模式。消息传递编程模型 MPI 已经被绝大多数并行计算环境支持，成为当今最流行的并行程序设计模式。

近些年，国际上对并行数值软件的研究进展非常迅速，已经完成了大量的并行数值软件包，如：PETSc、ScaLAPACK、TAO、UG 等。这些软件在各个高新技术领域发挥着重要的作用，它们使很多科研工作者和应用工程师从复杂、繁重的并行程序设计、调试中解脱出来。这些软件包大多由不同领域的科学家合作完

成，集成了经典、成熟的数值计算方法，并且经过了底层的性能优化，还具有面向各个应用领域的接口。

并行数值软件的研究和开发是一个包括数学、计算机、软件工程等多学科的交叉研究领域。并行数值软件的研究和开发要比串行数值软件困难得多，它要求开发者不仅拥有数学、物理等方面的理论知识，掌握软件设计、开发，并行编程、调试、性能优化等技术，而且还必须熟悉各种并行计算环境，了解国际上最新并行数值软件的发展动态。总的来说，并行数值软件的研究与开发是非常具有挑战性的工作。

目前流行的与本研究相关的并行数值软件包主要有

(1) **PETSc** [14]：PETSc 是美国 Argonne 国家重点实验室数学与计算机部的一组研究人员 Satish Balay, William Gropp, Lois C. McInnes, Barry Smith 等及相关访问学者于 1995 年开始开发的可移植可扩展科学计算工具箱。PETSc 采用面向对象程序设计，具有很强的可移植性、可扩展性；它面向偏微分方程的科学计算，具有多种科学计算标准组件，如线性方程组解法器、非线性方程组解法器、偏微分方程时间步解法器 (TS) 等；同时，PETSc 具有完整的文档和优质的维护。PETSc 逐渐成为受人瞩目并得到大量应用的一种科学计算工具。

(2) **PadFEM** [15]：PadFEM 是一个支持并行自适应有限元模拟的平台。除了基于 Tcl/Tk 和 Java 的图形预处理接口外，还包括对二维任意区域的基于四叉树的网格生成器、网格划分算法、各种解法器、误差估计器以及网格细化和负载平衡算法。对于二维监控、调试和后处理，提供了基于 Xlib 的图形接口 XFem。并行化模式为基于网格划分的 SPMD。该软件允许求解定常和非定常具有混合边界条件 (Dirichlet Neumann 类) 的 Poisson 方程，及非定常 Navier-Stokes 方程。

(3) **UG** [18]：UG 的开发始于 1990 年 7 月。如今大约十人正工作于其核心功能之上。当前版本具有大约 450,000 行的 C 代码。UG 是一个在二维和三维空间无结构网格上使用自适应多重网格方法的灵活的数值求解偏微分方程的软件工具。其设计思想灵活多样，支持多种离散格式，适用于多种不同背景的应用。UG 内核程序设计为与待求解的偏微分方程无关的。它提供几何和代数数据结构及许多网格处理方法、数值算法、可视化技术和用户界面。UG 支持多种单元类型，采用局部层次细化和粗化方法实现自适应。实现了 CG、CR、BiCGSTAB、乘法局部多重网格、ILU、Gauss-Seidel、Jacobi 和 SOR 等解法器。基于最新版 UG 内核程序的问题类包括：标量对流扩散、非

线性扩散、线弹性、弹塑性、不可压缩、多孔渗流中密度驱动流和多相流。但是 UG 过于庞大的代码量使继续开发以及代码维护极其困难。

(4) **DAGH** [17]：DAGH 是为 NSF 重大挑战项目 Binary Black Hole 而开发的一个计算工具箱。它提供了一个用自适应有限差分求解偏微分方程组的框架，可用于求解二维和三维问题。计算过程可根据用户的需求采用串行或并行模式完成。DAGH 提供用户程序接口，允许用户用传统的 FORTRAN 77 和 Fortran 90 或 C 和 C++ 来实现具体的数值计算过程。DAGH 为实现使用自适应网格细化算法求解偏微分方程提供一个程序开发平台。分级提取 (hierarchical abstraction) 和内容的分离 (separation of concerns) 被集成到 DAGH 的开发之中。在自适应求解过程中，从一个基础粗网格开始。在求解过程中，用刻画解的部分参数，如局部截断误差，来识别需要进一步求解的区域。仅在这些区域添加更细的子网格，将这一过程递归地做下去直到达到一个最大细化层或局部截断误差已经降到期望的水平以下。因此，在自适应网格细化计算中网格步长仅对基础网格是固定的，对子网格依问题的需要来局部地确定。该软件主要是针对有限差分格式的。

(5) **ParMETIS** [12] [13]：由 George Karypis, Kirk Schloegel 和 CipinKumar 开发的 ParMETIS (parallel graph partitioning and sparse matrix ordering library) 是一个基于 MPI 的并行软件包，其中的算法主要涉及两方面：

- 剖分和重新剖分无结构图
- 压缩和重排稀疏矩阵

ParMETIS 特别适合并行数值模拟中的网格剖分。在对网格剖分的过程中，ParMETIS 能够在保持各子区域大致平衡的前提下最大限度地减少内边界上的单元数目，从而节省并行计算中 MPI 通信的开销。ParMETIS 的核心算法来源于 METIS，但是做了大量的扩展和延伸，能够处理分布式存储的数据，做到了真正意义上的并行。与 METIS 相比，ParMETIS 更适合于并行计算和大规模数值模拟。目前，ParMETIS 主要提供以下功能：

- 剖分无结构图和矩阵
- 重分自适应细化图
- 剖分多重相位和多重物理模拟图
- 改进现有剖分结果
- 计算稀疏矩阵的压缩存储

- 构建网格的对偶图

所有 ParMETIS 的函数都以下面所介绍的图的特征作为输入参数：

- 图的邻接结构，也就是图中顶点之间的相对关系。
- 图中顶点和边上的权重。
- 图在各处理器上的分布情况。

在不同的应用环境下，ParMETIS 所处理的图的含义也不尽相同。当进行网格剖分时，ParMETIS 首先将网格对偶到一个无向图，图中顶点和边的含义可以根据需要来选定。例如，可以将对偶图中的每个顶点定义为网格中的一个单元；如果两个单元是相邻的，那么在对偶图中，代表这两个单元的顶点就会有一条边相连，相邻单元的定义可以采用拥有公共边、拥有公共面、拥有公共顶点等多种方式。

上面只是简单列举了目前已有的一些并行软件包，每一种都有其各自的优缺点，使用时需要根据所求解问题的特点及计算环境进行选用。

本文的工作围绕我们自主开发的基于消息传递模型的并行自适应有限元平台 PHG 进行，在该平台中实现了线性棱单元，并利用线性棱单元对三维时谐场问题进行了大型并行数值模拟实验。

§1.3 Maxwell 方程组

电磁学是物理学中的重要领域，电磁分析问题主要是求解给定边界条件下的麦克斯韦（Maxwell）方程组。本文的数值实验将使用 PHG 的线性棱单元来求解麦克斯韦方程组，因此这一节中简要介绍一下麦克斯韦方程组的基本知识。

麦克斯韦方程组刻划了宏观电磁现象的本质规律，对于基本的电磁场变量，方程组包含以下四个一阶偏微分方程：

$$\operatorname{curl}\vec{E}+\frac{\partial B}{\partial t}=0 \qquad \text{（法拉第定律）} \tag{1.1}$$

$$\operatorname{curl}\vec{H}-\frac{\partial D}{\partial t}=J \qquad \text{（麦克斯韦 - 安培定律）} \tag{1.2}$$

$$\nabla\cdot\vec{D}=\rho \qquad \text{（高斯定律）} \tag{1.3}$$

$$\nabla\cdot\vec{B}=0 \qquad \text{（磁场高斯定律）} \tag{1.4}$$

其中：

$\vec{E}$ = 电场强度 (伏特/米)

$\vec{D}$ = 电通量密度 (库仑/米^2)

$\vec{H}$ = 磁场强度 (安培/米)

$\vec{B}$ = 磁通量密度 (韦伯/米^2)

$\vec{J}$ = 电流密度 (安培/米^2)

ρ = 电荷密度 (库仑/米^3)

另一个基本方程是连续性方程，可以写成

$$\nabla\cdot\vec{J} = -\frac{\partial\rho}{\partial t} \tag{1.5}$$

它表示电荷守恒。

方程（1.1）–（1.5）中只有三个是独立的，称为独立方程。前三个方程，方程（1.1）–（1.3），或者前两个方程，方程（1.1）和（1.2），以及方程（1.5），都可以被选作独立方程。其它两个方程，方程（1.4）和（1.5）或者方程（1.4）和（1.3）可由独立方程导出，因此被称为辅助方程或相关方程。

由于五个麦克斯韦方程中只有三个是独立的，方程数小于未知量个数，所以三个独立方程是非定解的形式。当场量间的本构关系确定后，麦克斯韦方程组才有定解形式。本构关系描述了被考虑介质的宏观性质。对于简单介质，他们是

$$\vec{D} = \varepsilon\vec{E} \tag{1.6}$$

$$\vec{B} = \mu\vec{H} \tag{1.7}$$

$$J = \sigma\vec{E} \tag{1.8}$$

其中，本构参数 ε、μ 和 σ 分别表示介质的介电常数、磁导率和电导率。对各向异性的介质，这些参数是张量；对各向同性的介质，它们是标量。对于非均匀介质，它们是位置的函数；对均匀介质，它们不随位置变化。本文主要考虑介质为线性、各向同性的情况，此时，ε、μ 和 σ 是关于空间变量 x 的有界标量函数。联立上述方程，即得到描述电磁波与介质之间相互作用的二阶双曲型系统。

在不同类型的电磁场问题中，往往可以针对电磁场中的时间变量提出一些假定以简化系统，得到相应的数学模型。本文研究的情况是假设场量是单频谐振函数时的时谐场。这时可将场量分解为：$\vec{E}(x,t) = \mathrm{Re}(\hat{E}(x)e^{i\omega t})$ 和 $\vec{H}(x,t) = \mathrm{Re}(\hat{H}(x)e^{i\omega t})$，从而麦克斯韦方程组简化为频域上的不定二阶偏微分方程。这种时谐场模型可以用于计算散射问题，波导问题 [22] [23] 和传播模式，最近还被用于求解本征值问题 [24]。

要确定电磁场，还必须给出求解域的边界条件。边界条件是多种多样，因问题而异的。在电磁领域中，很多闭区域问题的求解域边界都是金属，如腔本征值问题，金属体的散射问题等。如果视金属为理想导体，那么此类问题的边界条件就可以写成

$$\vec{n} \times \vec{E} = \vec{g} \tag{1.9}$$

或

$$\vec{n} \times \operatorname{curl} H = \vec{g} \tag{1.10}$$

这里 $\vec{n}$ 是边界的单位法向量。数学上称式 (1.9) 为第一类边界条件，其特征为未知量在边界为已知固定值；式 (1.10) 为第二类边界条件，其特征为未知量的导数在边界为已知固定值。

与闭区域不同，开区域问题（如幅射和散射问题）的边界条件通常不能写成第一或第二类边界条件，而是第三类边界条件。这类边界的特征是未知量和未知量的导数在边界有确定的关系。如自由空间的幅射和散射问题，其无穷远处的边界条件为：

$$\lim_{r\to\infty} \left[\operatorname{curl} \begin{pmatrix} E \\ H \end{pmatrix} + jk_0 \hat{r} \times \begin{pmatrix} E \\ H \end{pmatrix} \right] = 0 \tag{1.11}$$

其中 $\hat{r} = \sqrt{x^2 + y^2 + z^2}$。这也就是通常所说的索末菲幅射条件。本文中的算例将主要处理闭区域的边界条件。

§1.4 论文的主要工作

本论文的主要工作是在三维自适应有限元软件平台 PHG 上实现线性棱单元，以及针对棱单元计算的需要测试、改进 PHG 中的相关数据结构和代码。

PHG 为有限元计算设计了一组高层的数据结构和操作，主要通过自由度对象及相关的操作来实现。这些数据结构及操作以一种通用的、面向对象的方式封装了复杂的网格操作、数据分布、并行通信等细节，并将这些部分对用户完全隐藏起来，使得用户代码不必关心与网格、并行等相关的内容，做到与串行程序一样简洁。同时，这样的设计思想也方便了新的有限元类型在 PHG 中的实现。例如，当添加一种新的有限元类型 (基函数) 时，只要定义好相应的自由度类型，提供该类型所需的几个基本操作，包括基函数及其梯度值的计算公式、函数空间到有限元空间的插值操作 (即自由度对象的初始化)、标准单元上父子单元间的插值关系，就可以使用 PHG 中已经实现的所有高层操作，如与基函数相关的数值积分、有限元函数的各种运算 (包括代数、微分、积分运算)、以及当网格变化时自由度变量的插值、迁移等。由于最初的 PHG 代码中只实现了标量类型，主要是

Lagrange 类的有限元基函数，一些当初设计的数据结构、函数接口并不完全适合于矢量类型的基函数，一些函数中的代码处理矢量类型的基函数时有错误。本文的主要目的就是通过线性棱单元的实现来检验、完善 PHG 中的相关数据结构以及代码中与矢量基函数相关的操作，并通过电磁场问题的数值模拟来测试相关代码的正确性及计算效率。具体地，论文主要完成了下述几方面的工作：

(1) 在 PHG 中实现了线性棱单元的自由度类型。

(2) 修改 PHG 中的模块使之适合于处理矢量基函数，并修正了一些相关的数据结构和函数接口。对 PHG 的各种高层操作进行了大量测试，保证了这些操作的正确性，为将来高阶棱单元的实现打下了基础。

(3) 改进了 PHG 的数值积分函数的基函数 cache 机制，使其接口更加规范，适用性更强。

(4) 修改了 PHG 的线性解法器接口，使之能够处理以多个自由度对象为未知量的情况。

(5) 在此基础上实现了三维时谐 Maxwell 方程的并行自适应有限元计算程序，并进行了大型并行数值模拟实验。

PHG 是我们完全自主设计的一个基于网格二分细化、适合于分布式存储并行计算机的三维并行自适应有限元平台。由于目前国际上尚没有公开发布的类似软件，论文中的工作所涉及的许多研究内容是探索性、创新性的。虽然 PHG 尚处在初始开发阶段，但其核心部分的源代码亦已超过 3 万行，对其底层数据结构、函数接口的一个修改往往涉及到多个文件、大量代码的改动及随之而来的大量测试工作。另外，原来的代码中，与矢量基函数相关的诸多部分没有经过测试，并且有些是错的，有时需要通过大量实验、分析才能找出原来代码中存在的问题。这些因素增加了本论文工作的难度。

§1.5 论文内容安排

包括本章在内，本论文共分为五章。第一章为引言，包含对自适应有限元方法、Maxwell 方程组背景知识的回顾和现有并行数值软件概况的介绍。第二章对正在开发中的并行自适应有限元平台 PHG（Parallel Hierarchical Grid）进行了介绍，并讨论了数值积分函数接口的设计及本文对基函数及导数值计算中的 cache 机制的改进。数值积分函数中对基函数及导数值的 cache 机制在保证计算效率的

情况下简化了用户接口。第三章介绍线性棱单元的背景以及具体实现过程，包括线性棱单元赋值函数、基函数等函数接口的实现细节。第四章介绍了利用线性棱单元使用自适应有限元方法对三维时谐场问题在并行计算机上进行的数值模拟实验结果，这些数值实验结果验证了本文所实现的线性棱单元及相关代码的正确性和有效性，我们的程序运行稳定，对各种问题和处理器、网格规模均表现出很好的的鲁棒性，并且在并行效率、并行可扩展性方面也取得了令人满意的结果。

第二章　PHG 介绍

PHG（Parallel Hierarchical Grid）是一个三维并行自适应有限元软件平台。其核心是分布式的层次网格结构。PHG 目前处理的网格对象是三维四面体协调网格。PHG 的并行实现基于 MPI [9]。

PHG 采用网格单元二分局部细化算法，也称为“边细化”算法。对一个单元细化时，将它的一条边（称为*细化边*）的中点和与之相对的两个顶点相连，使该单元一分为二，产生两个新的四面体单元，该单元被称为细化所产生的两个新单元的*父亲单元*，而细化所产生的新单元则称为它的*子单元*。PHG 中采用的细化算法基于 [25, 28, 26, 27]。单元细化边的标注借用了自适应有限元软件包 ALBERTA [16] 中的记号，PHG 中对它进行了扩展以支持任意的四面体初始网格，实现单元细化边及细化类型的自动生成。初始网格中，单元的类型和顶点编号可以由用户指定，也可以由 PHG 自动产生。

PHG 提供一套灵活的自由度管理机制，以及统一的解法器接口。PHG 支持将网格及自由度以 VTK 的格式输出到指定文件以便用其它基于 VTK 的可视化软件如 ParaView [19]，MayaVi [20] 等处理。

本章 §2.1 节将简单介绍 PHG 中基本数据结构，§2.2 节将介绍自由度管理，§2.3 节将对解法器接口进行简单的介绍，§2.4 节介绍数值积分函数接口及其中的 cache 机制。

§2.1　基本概念及数据结构

§2.1.1　单元内编号、本地编号和全局编号

当用 p 个进程进行并行处理时，PHG 将网格剖分成 p 个非重叠的子网格。因此 PHG 中的几何对象，包括顶点、边、面、单元、自由度等，有三种不同的编号方式，分别称为*单元内编号*、*本地编号*和*全局编号*。单元内编号指在对象所在的单元内的编号，本地编号又称为*子网格内编号*，它指对象在当前子网格中的编号，而全局编号则指对象在全局网格中的编号。

§2.1.2　SIMPLEX

PHG 描述单元的数据结构是 `SIMPLEX`，其中包含如下几个主要成员：

```
typedef struct SIMPLEX_ {
    struct SIMPLEX_     *children[2];
    void                *neighbours[NFace];
```

```
    void                *parent;
    INT                 verts[NVert];
    INT                 edges[NEdge];
    INT                 faces[NFace];
    INT                 index;
    SHORT               mark;
    BTYPE               bound_type[NFace];
    ... ...
} SIMPLEX;
```

其中，children[0] 和 children[1] 分别指向两个子单元；neighbours[i] 指向第 i 个面上的邻居单元，如果面 i 为边界面则 neighbours[i] 为空指针，如果面 i 上的邻居不在本地，则 neighbours[i] 指向的是描述子网格间邻居关系的一个结构 (普通用户通常不必关心)；相应地，bound_type[i] 给出面 i 的边界条件类型，可取值为 INTERIOR (内部面)、DIRICHLET (Dirichlet 边界面)、NEUMANN (Neumann 边界面)、UNDEFINED (未指定类型的边界面) 和 REMOTE (邻居在其它子网格中的内部面，亦即子网格的内边界面) 按二进制位位的组合；parent 指向父亲单元，对根单元而言 parent 为空指针。

verts[]、edges[]、faces[] 和 index 成员分别保存顶点、边、面和单元的本地编号，它们的全局编号可以分别调用宏 GlobalVertex、GlobalEdge、GlobalFace 和 GlobalElement 获得。例如，假设 g 为指向当前网格的指针，e 为指向一个单元的指针，则 e->verts[0] 给出单元 e 中第 0 个顶点的本地编号，而

```
GlobalVertex(g, e->verts[0])
```

则给出该顶点的全局编号 (对于非分布式的网格本地编号与全局编号是一样的)。

mark 成员用于在自适应计算中标注希望细化或粗化的单元，mark > 0 表示要求将该单元细化 mark 次，mark < 0 表示允许将该单元最多粗化 $-$mark 次。

§2.1.3 GRID

PHG 描述网格的数据结构是 GRID，其中包含如下几个主要成员：

```
typedef struct GRID_ {
    FLOAT       lif;
    COORD       *verts;
    ... ...
    INT         nleaf;
```

```
    INT         nvert;
    INT         nedge;
    INT         nface;
    INT         nelem;
    INT         nvert_global;
    INT         nedge_global;
    INT         nface_global;
    INT         nelem_global;
    INT         nroot;
    INT         ntree;
    ... ...
    int         rank;
    int         nprocs;
#if USE_MPI
    MPI_Comm    g->comm;
#endif
    ... ...
} GRID;
```

这里重点解释一下实现棱单元涉及到的 verts 和 nxxxx、nxxxx_global 成员。

verts 数组用于保存子网格中所有顶点的迪卡尔坐标，按顶点的本地编号顺序存放。例如，假设 g 为网格指针，e 为单元指针，则 e 的第 i 个顶点的 x、y、z 坐标分别为：

```
g->verts[e->verts[i]][0]
g->verts[e->verts[i]][1]
g->verts[e->verts[i]][2]
```

nleaf 给出当前子网格包含的单元数，即当前子树包含的叶子单元数。

nvert_global、nedge_global、nface_global 和 nelem_global 分别给出当前全局网格中的顶点数、边数、面数和单元数。显然，nelem_global 等于所有子网格中的 nleaf 值之和。这些量在所有进程中完全一样。

§2.2 自由度对象

自由度对象是 PHG 的基本数据结构之一，用于描述分布在网格上的变量值。它既用于定义有限元函数空间，也用于存储、处理其它与网格相关的数据，如几

何信息 (单元的体积、面的面积和法向量等)。自由度对象包括两个基本的数据结构：DOF_TYPE 和 DOF。

§2.2.1 自由度类型

自由度类型描述自由度对象的基本特征，其数据结构部分主要成员如下：

```
typedef struct DOF_TYPE_ {
    ... ... /* caches */
    const char          *name;
    FLOAT               *points;
    struct DOF_TYPE_    *grad_type;
    ... ...
    /* 函数指针 */
    DOF_INTERP_FUNC     InterpC2F;
    DOF_INTERP_FUNC     InterpF2C;
    DOF_INIT_FUNC       Func;
    DOF_BASIS_FUNC      BasFuncs;
    DOF_BASIS_GRAD      BasGrads;
    ... ...
    BOOLEAN             invariant;
    SHORT               nbas;
    BYTE                order;
    BYTE                dim;
    ... ...
    BYTE                np_vert;
    BYTE                np_edge;
    BYTE                np_face;
    BYTE                np_elem;
} DOF_TYPE;
```

- name 给出自由度类型的名称或描述信息。
- grad_type 给出该自由度类型所定义的函数的梯度的自由度类型，用于自动生成函数的梯度、散度、旋度等自由度对象。
- np_vert、np_edge、np_face 和 np_elem 分别给出定义在顶点、边、面和体中

的自由度个数。

- nbas 给出一个单元中自由度或局部基函数的个数。
- invariant 说明自由度类型的局部基函数是否与单元形状无关。
- order 给出基函数的多项式次数，为选择数值积分精度提供自动判断依据。
- dim 给出基函数的维数。
- 数组 points 顺序给出顶点、边、面和单元自由度的位置信息，分别用 0 维、1 维、2 维和 3 维重心坐标表示。
- InterpC2F、InterpF2C 是插值函数，具体解释见 §3.2.4。
- Func 是自由度赋值函数，其接口类型为 DOF_INIT_FUNC，具体如下：

```
void Func(const DOF *dof, const SIMPLEX *e, GTYPE type, int index,
          DOF_USER_FUNC userfunc,
          DOF_USER_FUNC_LAMBDA userfunc_lambda,
          FLOAT *funcvalues, FLOAT *values)
```

  具体解释见 §3.2.3。

- 函数 BasFuncs 返回给定重心坐标位置的基函数值，函数接口类型为 DOF_BASIS_FUNC，接口参数如下：

```
FLOAT *BasFuncs(GRID *g, SIMPLEX *e, int no, FLOAT *lambda)
```

  其中 no 为局部基函数编号 (依次按照单元中顶点、边、面和体自由度的顺序编号，从 0 开始)，调用时如果 no < 0 则表示要求计算所有基函数的值。lambda[Dim + 1] 给出重心坐标。函数返回一个缓冲区地址，其中包含所指定或全部基函数的值，该缓冲区由 BasFuncs 提供。具体解释见 §3.2.1。

- BasGrads 返回给定重心坐标位置的基函数关于重心坐标的梯度的值，接口类型为 DOF_BASIS_GRAD，接口参数为：

```
FLOAT *BasGrads(GRID *g, SIMPLEX *e, int no, FLOAT *lambda)
```

  各参数的含义与 BasFuncs 类似。具体解释见 §3.2.2。

实现线性棱单元关键就在于定义一种线性棱单元的自由度类型，对自由度类型中的成员赋值，并实现相应的接口函数，第三章 将会作详细的介绍。

§2.2.2 自由度对象数据结构

自由度对象数据结构中的部分主要成员如下：

```
typedef struct DOF_ {
    char        *name;
    GRID        *g;
    DOF_TYPE    *type;
    FLOAT       *data;
    ... ...
    ... ...
    ... ...
    BYTE        dim;
} DOF;
```

一个自由度对象必须与一个网格相关联。一个网格对象中记录着所有与它相关联的自由度对象，当网格改变时 (如细化、粗化、重分布)，这些自由度对象会被自动更新，而当网格释放时，所有自由度对象会被自动释放。

`dim` 成员给出自由度对象的维数。一个自由度对象所对应的函数维数等于自由度类型的维数与自由度对象的维数之积 (`dim` × `type->dim`)。

`data` 指向存储自由度对象的数据的缓冲区，它的长度等于自由度对象在当前网格中的所有自由度个数；每个进程只保存属于自己的子网格上的自由度数据。对于同时属于多个子网格的顶点、边或面自由度，它们重复存储在不同子进程中。自由度对象的数据依照顶点自由度、边自由度、面自由度和体自由度的顺序根据它们各自的本地编号顺序连续存放，当网格变化时 PHG 会自动对它们进行调整。

自由度对象的数据结构使得我们可以对它们进行各种代数、微分、积分运算，如梯度、散度、旋度、L^2 模等运算。

§2.2.3 几何量自由度对象

在进行有限元计算时，常常要用到单元的几何量，包括单元面的面积、法向量、直径，单元的体积、直径和重心坐标的 Jacobi 矩阵等。为了使用及管理方便，PHG 中定义了一个特殊自由度对象用于存储这些几何量，并提供了一系列函数接口用于获取这些几何量，它们分别是：

- 获取网格 `g` 中指定单元 `e` 的体积。

```
FLOAT phgGeomGetElement(GRID *g, SIMPLEX *e)
```

- 获取网格 g 中指定单元 e 的直径。

```
FLOAT phgGeomGetDiameter(GRID *g, SIMPLEX *e)
```

- 获取网格 g 中指定单元 e 的重心坐标的 Jacobi 矩阵。

```
FLOAT *phgGeomGetJacobian(GRID *g, SIMPLEX *e)
```

- 获取网格 g 中单元 e 上指定面的面积，face 为面的单元编号。

```
FLOAT phgGeomGetFaceArea(GRID *g, SIMPLEX *e, int face)
```

- 获取网格 g 中单元 e 上指定面的直径，face 为面的单元编号。

```
FLOAT phgGeomGetFaceDiameter(GRID *g, SIMPLEX *e, int face)
```

- 获取网格 g 中单元 e 上指定面的单位外法向量，face 为面的单元编号。

```
FLOAT *phgGeomGetFaceOutNormal(GRID *g, SIMPLEX *e, int face)
```

§2.3 解法器

PHG 中的线性解法器接口目前还是初步的，只能满足求解线性 PDE 的要求，拟在将来开发过程中根据应用需求逐步扩展。PHG 为求解线性方程组定义了一个 SOLVER 对象，并提供了若干接口函数。目前 PHG 提供了对四种解法器的支持，它们分别是 PETSc、SuperLU_dist、LASPack 和 SPC，其中 LASPack 是一个串行解法器，主要供在没有 MPI 的机器上使用，其余三个均为并行解法器。PHG 提供的常用解法器接口函数有：

- 创建解法器对象：

```
SOLVER *phgSolverCreate(OEM_SOLVER *oem_solver, DOF *u, ...)
```

- 销毁解法器对象，释放占用的资源：

```
int phgSolverDestroy(SOLVER *solver)
```

- 累加线性方程组系数矩阵：

```
int phgSolverAddMatrixEntry(SOLVER *solver, INT row, INT col,
              FLOAT value)
int phgSolverAddMatrixEntries(SOLVER *solver,
              INT nrows, INT *rows, INT ncols, INT *cols,
              FLOAT *values)
```

- 累加线性方程组右端项：

```
int phgSolverAddRHSEntry(SOLVER *solver, INT index, FLOAT value)
int phgSolverAddRHSEntries(SOLVER *solver, INT n, INT *indices,
              FLOAT *values)
```

- 求解线性方程组：

```
int phgSolverSolve(SOLVER *solver, BOOLEAN destroy, DOF *u, ...)
```

§2.4 数值积分

有限元计算中，计算单元刚度矩阵以及右端项需要计算单元基函数或其梯度、旋度等的积分。例如对于下面的简单模型问题 (Poisson 方程)：

$$\begin{cases} -\Delta u = f & x \in \Omega \\ u = g & x \in \partial\Omega \end{cases} \tag{2.1}$$

我们在计算单元刚度矩阵时, 需要计算积分：

$$\iiint_e \nabla W_i \cdot \nabla W_j \,\mathrm{d}V, \quad i,j = 1,2,\ldots,\texttt{nbas} \tag{2.2}$$

PHG 提供了采用具有最高代数精度的 Gauss 型数值积分方法 [34] [35] [37] [38] [39] 的数值积分函数来完成这些积分。

§2.4.1 数据结构

Gauss 型数值积分最重要的部分是 Gauss 积分点的选取及其权重的确定。对于不同精度的高斯积分，Gauss 积分点的数量是不一样的。类似于 ALBERTA [16] 中的 QUAD 结构，我们也定义了一个结构体，尽可能的将不同阶积分所需要的信息封装在结构体中，我们称之为积分子（也叫积分类型）。具体定义如下：

```
typedef struct QUAD_ {
    const char  *name;
    int         dim;
    int         order;
    int         npoints;
    FLOAT       *points;
    FLOAT       *weights;
    SHORT       id;
} QUAD;
```

结构体成员说明:

- `name`: 定义积分子的名称，可以由用户自己指定，如三维的四阶精度的积分可以命名为 `"3D P4"`。
- `dim`: 定义积分子的维数，即 1 维代表线段上的积分，2 维代表三角形上的积分，3 维代表四面体上的积分。
- `order`: 定义积分子的阶数。
- `npoints`: 定义该积分子需要的 Gauss 积分点的个数。
- `points`: 定义该积分子需要的 Gauss 积分点在单元中的重心坐标。
- `weights`: 定义每个 Gauss 积分点的权重。
- `id`: 定义积分子的一个标识，与用户无关，初值为 -1，在 §2.4.3 节中我们会作更详细的介绍。

目前 PHG 预定义了 1–9 阶一维（线段上）积分子，1–9 阶二维（三角形中）积分子和 1–9 阶三维（四面体中）积分子，它们的变量名为 `QUAD_[123]D_P[1-9]`，采用的是 Legendre-Gauss 积分公式 [35]，个别高维积分公式是利用一维 Legendre-Jacobi 积分公式通过张量积构造的。除了预定义的积分子，用户可以自定义更高阶或者其它类型的 Gauss 求积公式。例如，3 阶的 Radau 积分公式 [36] 可以如下定义

```
static FLOAT QUAD_1D_Radau_pts[ ]= {0, 1./6.};
static FLOAT QUAD_1D_Radau_wts[ ]= {.5, 1.5};
QUAD QUAD_1D_Radau3_ = {
```

```
    “1D Radau1”,
    1,
    3,
    2,
    QUAD_1D_Radau_pts,
    QUAD_1D_Radau_wts,
    -1
};
#define QUAD_1D_Radau3 (&QUAD_1D_Radau3_)
```

还有一个特殊的积分子 QUAD_DEFAULT，其定义为 (QUAD *)NULL，它告诉接口函数根据基函数的次数自动选取适当的积分子。

§2.4.2 常用接口函数

积分子只是定义了积分类型。用户只有通过使用如下的一些接口函数才能进行积分计算。为了便于介绍，我们约定:

- 函数 U 和 V 对应的自由度对象分别是 u 和 v；$\Pi_h U$ 和 $\Pi_h V$ 分别代表 U 和 V 在有限元空间上的投影（插值）。
- W_n 代表自由度 u 在单元上的第 n 个基函数。
- M_m 代表自由度 v 在单元上的第 m 个基函数。
- $[\cdot]$ 代表面上的跳量。例如 $[\Pi_h U]$ 代表不连续函数 $\Pi_h U$ 在相邻单元的邻接面上的变化量。

常用的接口函数如下：

```
FLOAT phgQuadDofDotDof(SIMPLEX *e, DOF *u, DOF *v, QUAD *quad)
```

函数说明:

该函数计算 u 和 v 在单元 e 上的内积。可选参数 quad 用于指定积分子，如果将它取为 QUAD_DEFAULT，则函数将根据自由度 u 的类型自动选择适当精度的积分子。确切地，该函数计算并返回 $\iiint_e \Pi_h U \cdot \Pi_h V$。

```
FLOAT phgQuadBasDotBas(SIMPLEX *e, DOF *u, int n, DOF *v, int m,
                       QUAD *quad);
```

函数说明:

该函数计算单元 e 上 u 的第 n 个局部基函数与 v 的第 m 个局部基函数的内积。quad 参数同 phgQuadDofDotDof。确切地，该函数计算并返回 $\iiint_e W_n \cdot M_m$。

```
FLOAT phgQuadGradBasDotGradBas(SIMPLEX *e, DOF *u, int n, DOF *v,
                               int m, QUAD *quad)
```

函数说明:

该函数计算 u 与 v 的局部基函数的梯度在单元 e 上的内积。各项参数同 phgQuadBasDotBas。确切地，该函数计算并返回 $\iiint_e \nabla W_n \cdot \nabla M_m$。

```
FLOAT phgQuadCurlBasDotCurlBas (SIMPLEX *e, DOF *u, int n,
                                DOF *v, int m, QUAD *quad);
```

函数说明:

该函数计算 u 与 v 的局部基函数的 curl 在单元 e 上的内积。各项参数同 phgQuadBasDotBas。确切地，该函数计算并返回 $\iiint_e \operatorname{curl} W_n \cdot \operatorname{curl} M_m$。

```
FLOAT *phgQuadDofTimesBas(SIMPLEX *e, DOF *u, DOF *v, int n,
                          QUAD *quad, FLOAT *res)
```

函数说明:

该函数计算函数 u 与函数 v 的局部基函数的乘积在单元 e 上的积分。n 给出 v 的基函数编号，计算结果放在 res 指定的缓冲区并返回缓冲区地址 (当 u 对应的是向量函数时计算结果是向量值)。缓冲区由用户自己负责分配与释放。quad 参数同 phgQuadDofDotDof。确切地，该函数计算并返回 $\iiint_e \Pi_h U \cdot M_n$。

```
FLOAT phgQuadDofDotBas (SIMPLEX *e, DOF *u, DOF *v, int n,
                        QUAD * quad);
```

函数说明:

该函数计算函数 u 与函数 v 的局部基函数在单元 e 上的内积，它要求 DOF v 的基函数是矢量函数，这是与 phgQuadDofTimesBas 不同之处。确切地，该函数计算并返回 $\iiint_e \Pi_h U \cdot M_n$。

```
FLOAT phgQuadFuncDotBas (SIMPLEX *e, DOF_USER_FUNC f,
                         DOF *u, int n, QUAD *quad);
```

函数说明:

该函数计算解析函数 f 与有限元函数 u 的局部基函数在单元 e 上的内积。f 是由用户自定义的函数，例如，f(x，y，z，res) 表示通过 res 返回函数 f 在

点 (x，y，z) 处的值。res 的维数必须与 u 的基函数维数相匹配。quad 参数同 phgQuadDofDotDof。确切地，该函数计算并返回 $\iiint_e f \cdot M_n$。

```
DOF *phgQuadFaceJump(DOF *u, DOF_OP proj, const char *name,
                     QUAD *quad)
```

函数说明:

该函数计算指定 DOF 对象 u 在网格面上的跳量的积分，通常用于计算有限元后验误差估计。它返回的是一个新的 DOF 对象，每个面上一个自由度，自由度的值是 u 在该面上投影以后的跳量的平方的积分，即 $\iint_f$ [$\Pi_h U$在面上的投影] · [$\Pi_h U$在面上的投影]。quad 参数同 phgQuadDofDotDof。参数 proj 指定如何对 u 进行投影，可以取下面一些值：

- DOF_OP_NONE：表示不投影；
- DOF_OP_DOT：表示与面单位法向量进行内积；
- DOF_OP_CROSS：表示与面单位法向量进行外积 (叉乘)。

通过这些接口函数，我们可以非常方便的计算单元刚度矩阵以及右端项。

§2.4.3 数值积分函数中的 cache 机制

通常，用户计算单元刚度矩阵时是以单元为对象进行的，并且需要反复用到基函数及其导数在一些相同积分点上的值。对于一类基函数，如 Lagrange 型基函数，它们在给定积分点上的值在所有单元中是一样的。而对于另一类基函数如棱单元，基函数在同一积分点位置的值在不同单元是不同的。为了避免在数值积分计算中重复计算同一个基函数在同样位置的值，一些有限元软件单独提供了计算并保存基函数在 Gauss 积分点处的值的数据结构与接口函数，如 ALBERTA 中的 quad_fast 结构，用户程序在计算数值积分前预先调用它计算并保存基函数在一个单元中所有积分点位置的值，然后再利用这些基函数的值计算数值积分。这种设计使得用户程序的结构比较繁琐。

我们在 PHG 中设计了一种 cache 机制来解决这一问题，它将当前单元中所有计算过的基函数及其梯度值保存在一些内部缓冲区中，这些缓冲区是动态生成的并且记录在自由度类型对象的数据结构中，供所有基于同一自由度类型的自由度对象所共用，如图 2.1 所示。这种 cache 机制对用户是透明的，用户程序中只需要直接调用上节所介绍的数值积分函数即可，而不必关心如何保存并重复利用已经计算过的基函数及其梯度的值。

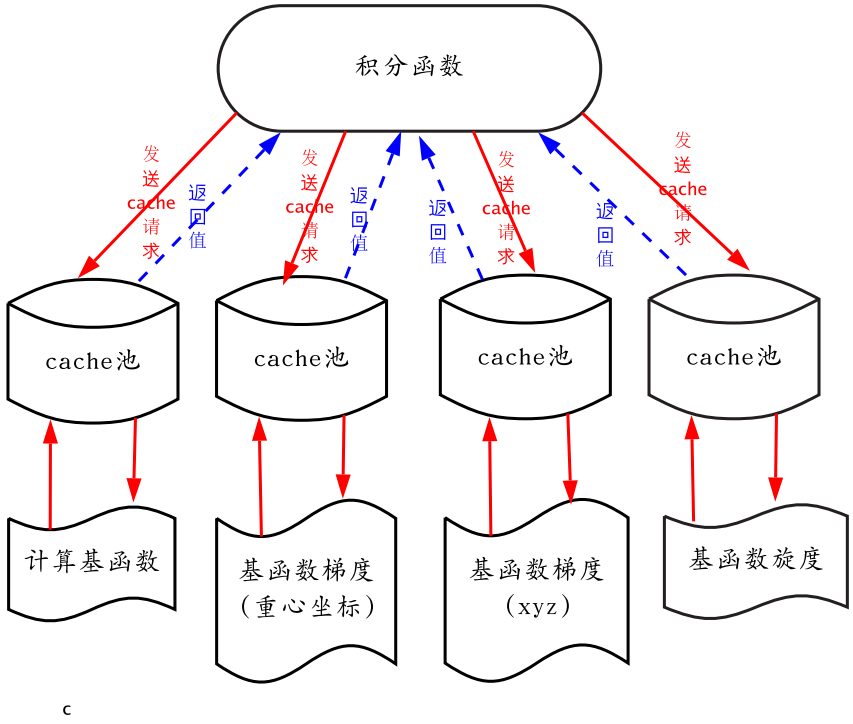

图 2.1 Cache 机制数据流图

为了实现这种 cache 机制，我们设计了两个内部数据结构 QUAD_CACHE 和 QUAD_CACHE_LIST 用于存储当前单元上的基函数或函数值；同时在 DOF_TYPE 和 QUAD 数据结构中添加了相应的成员项。

QUAD_CACHE 数据结构定义为：

```
typedef struct {
    SIMPLEX  *e;
    FLOAT    *data;
} QUAD_CACHE;
```

该数据结构主要用于存储缓存的具体数据，其中指针 e 指向缓存的数据对应的网格单元。指针 data 指向缓存数据的缓冲区。

QUAD_CACHE_LIST 数据结构定义为：

```
typedef struct {
    QUAD_CACHE  **caches;
    SHORT       n;
} QUAD_CACHE_LIST;
```

该数据结构在逻辑上可以理解为图 2.1 中的 cache 池。n 代表该池中能存储的 QUAD_CACHE 最大数量。

由于要缓存的数据都是和自由度基函数相关的，所以我们让 cache 池与自由度类型相关联。在 DOF_TYPE 结构中添加了如下一些成员项：

```
   ...
    void      *cache_basfunc;
    void      *cache_basgrad;
    void      *cache_gradient;
    void      *cache_curl;
    BOOLEAN   invariant;
   ...
```

其中，cache_basfunc、cache_basgrad、cache_gradient 和 cache_curl 是指向 QUAD_CACHE_LIST 对象的指针。invariant 变量为 TRUE 时说明该自由度类型的基函数在所有网格单元上是一样的；反之则当单元改变时需要重新计算。

在 QUAD 对象中我们定义了一个变量 id，该变量有两种状态：

- id = -1 时, 表明 cache 池中没有使用该积分子进行积分计算需要的相关数据，需要向 cache 池中添加记录。

- id > -1 时，表明使用该积分子进行积分计算需要的相关数据可能存在于 cache 池中（需要进一步判断），其位置为 id。

一个有效的 QUAD_CACHE 结构可以通过调用如下函数得到：

```
static inline QUAD_CACHE *
get_cache(void **clist_ptr, QUAD *quad)
```

函数说明:

输入参数为 QUAD_CACHE_LIST 的二重指针 clist_ptr 和 QUAD 的指针 quad。该函数返回一个 QUAD_CACHE 的指针。如果积分子 quad 已存在于 QUAD_CACHE_LIST 中，则返回的指针指向 QUAD_CACHE_LIST 中的某个 QUAD_CACHE；如果没有，则在 QUAD_CACHE_LIST 中分配新的空间，用于存储该 quad 类型积分对应的 QUAD_CACHE，新分配的 QUAD_CACHE 中的 data 为 NULL，并返回新分配的 QUAD_CACHE 地址。

以下四个函数调用 get_cache 函数获取 QUAD_CACHE 对象：

```
static FLOAT *
get_basfunc(SIMPLEX *e, DOF *u, int n, QUAD *quad)
```

函数说明:

输入参数为单元 e，自由度 u，基函数的序号 n，积分类型 quad。该函数返回自由度 u 在单元 e 上第 n 个基函数用于计算 quad 类型积分所需要的值。

```
static FLOAT *
get_basgrad(SIMPLEX *e, DOF *u, int n, QUAD *quad)
```

函数说明:

参数含义同 get_basfunc。返回值为基函数关于重心坐标的梯度。

```
static FLOAT *
get_gradient(SIMPLEX *e, DOF *u, int n, QUAD *quad)
```

函数说明:

参数含义同 get_basfunc。返回值为基函数关于笛卡尔坐标的梯度或者偏导数。

```
static FLOAT *
get_curl(SIMPLEX *e, DOF *u, int n, QUAD *quad)
```

函数说明:

参数含义同 get_basfunc。返回值为基函数的旋度。

当 get_cache 被调用时，它会返回一个指向 QUAD_CACHE 对象的指针。当遇到如下两种情况时 QUAD_CACHE 对象中缓存的数据是可用的，反之则不可用，必须重新计算。

(1) QUAD_CACHE 对象的 data 指针不为空而且该自由度类型的 invariant 标志为 TRUE, 即该自由度类型的基函数与单元无关；

(2) QUAD_CACHE 的 data 指针不为空而且 QUAD_CACHE 缓存的值刚好是单元 e 上的。

所有被使用了的积分类型都被保存在一个 QUAD 数组 quad_list 中。不同的缓存数据（如基函数的值、基函数梯度的值等）的 QUAD_CACHE 对象都被分别保存在不同的 QUAD_CACHE_LIST 中（与 DOF_TYEP 相关联）。目前 DOF_TYPE 中有 4 种 QUAD_CACHE_LIST，对于同一个单元，每个 QUAD_CACHE_LIST 可以缓存多种积分类型的数据。但是不能缓存同一积分类型在不同单元上的数据，如图 2.2 所示。图

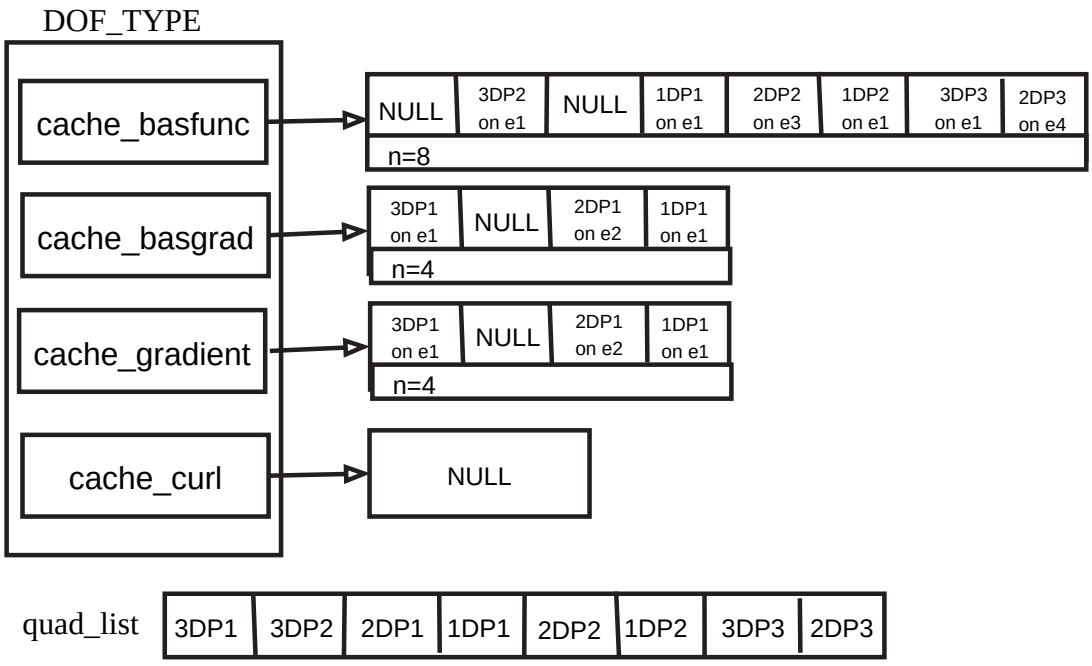

图 2.2 DOF_TYPE 中 cache 数据示例

中“3DP1”代表 3 维 1 阶积分，其它依此类推。

如果一个 QUAD 积分类型被缓存了，则 QUAD->id 代表该积分类型在 quad_list 以及相关 QUAD_CACHE_LIST 中的位置。这样在 QUAD_CACHE_LIST 中查找需要的

QUAD_CACHE 对象时，可以避免搜索，直接使用其 QUAD->id 定位。但是该做法可能会使 QUAD_CACHE_LIST 中产生“空洞”，如图 2.2 所示。这种 cache 结构的设计基于一个基本假设，即在有限元计算中，所涉及的不同基函数及积分类型是有限的，因此这些 cache 所占用的内存空间可以忽略不计，或者是可以容忍的。

最后我们给出 get_cache 函数的具体算法，

```
procedure get_cache(clist_ptr, quad)
   if(quad 未被缓存); then
       在 quad_list 中记录该 quad;
       给 quad->id 赋值;
   fi

   if clist_ptr is NULL; then
         初始化 clist_ptr;
   fi

   if clist->n <= quad->id; then
       将 clist->caches 的大小扩展到 quad->id + 1 项
       并且将 clist->caches[clist->n, ..., quad->id] 填充为 NULL;
       clist->n = quad->id;
   fi

   if clist->caches[quad->id] == NULL; then
       初始化 clist->caches[quad->id];
   fi
   return cache = clist->caches[quad->id];
```

§2.4.3.1 数值实验

为了验证 cache 机制的有效性，我们进行了一些数值实验。这些实验中分别使用一阶和三阶拉格朗日元求解 Poission 方程 (2.1)，以及使用线性棱单元求解时谐场问题。在合成刚度矩阵以及线性系统右端项的过程中，程序分别调用了：

- phgQuadGradBasDotGradBas
- phgQuadDofTimesBas

- `phgQuadCurlBasDotCurlBas`
- `phgQuadDofDotBas`
- `phgQuadBasDotBas`

我们对它们计算刚度矩阵的时间进行了统计（见表 2.1 表 2.2 表 2.3）。从表中我们发现使用 cache 所需时间明显比不使用 cache 所需时间少。以三阶拉格朗日元（表 2.2）最为明显，从表 2.2 中我们还能看出，即使计算规模达只达到 20449 个自由度 6144 个单元，不使用 cache 时生成刚度矩阵的时间也为 1109.6655 秒，而使用了 cache 后仅仅需要 2.5357 秒。而且从图 2.3 中可以看出，生成刚度矩阵需要的时间与单元数成正比，因此如果不使用 cache 机制，当计算规模扩大时生成刚度矩阵需要的时间将是不能忍受的。

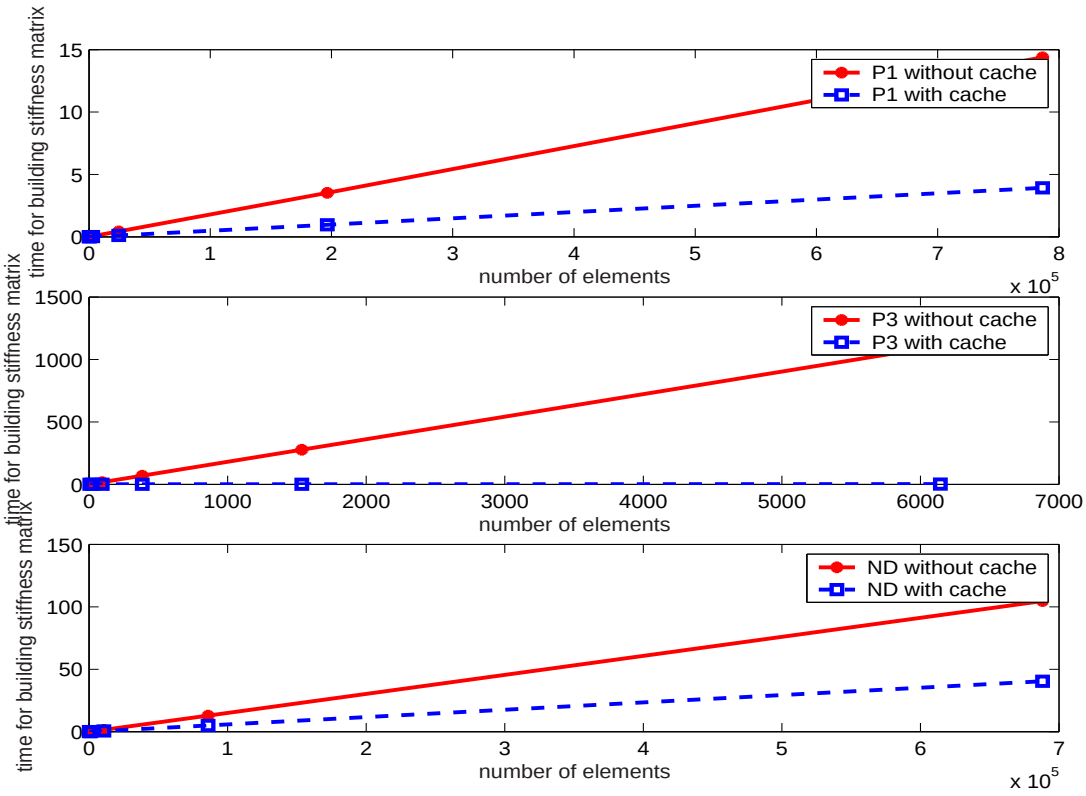

图 2.3 数值积分 cache 效率图

以上数值结果说明我们在 PHG 中实现的 cache 机制是有效的，它在保证程序计算效率的前题下简化了用户程序的接口。

表 2.1 使用一阶拉格朗日元合成刚度矩阵时间（单结点）

自由度/单元数	不使用 cache 时间（秒）	使用 cache 时间（秒）
8/6	0.0002	0.0001
27/48	0.0008	0.0002
25/384	0.0064	0.0015
729/3072	0.0523	0.0137
4913/24576	0.4366	0.1152
35937/196608	3.5228	0.9575
170081/786432	14.3831	3.9187

表 2.2 使用三阶拉格朗日元合成刚度矩阵时间（单结点）

自由度/单元数	不使用 cache 时间（秒）	使用 cache 时间（秒）
64/6	1.0800	0.0012
175/24	4.3267	0.0064
559/96	17.3195	0.0306
2197/384	69.2812	0.1293
8005/1536	277.3063	0.5857
20449/6144	1109.6655	2.5357

表 2.3 使用线性棱单元合成刚度矩阵的时间（单结点）

自由度/单元数	不使用 cache 时间（秒）	使用 cache 时间（秒）
281/168	0.0214	0.0084
1924/1344	0.1839	0.0701
14168/10752	1.5462	0.5950
108592/86016	12.8804	4.9415
850016/688128	104.6353	40.3481

第三章　线性棱单元

在电磁场有限元计算中，使用传统的节点有限元来表示矢量电场或磁场会遇到几个严重的困难 [42]：首先，出现非物理解即伪解，该解的产生是由于未强加散度条件所引起的；其次，不便于在材料界面或导体表面强加边界条件；最后，对区域的凹角和边缘处处理很困难。棱单元的出现，为描述场的变化和连续性提供了有效的物理框架，是对传统节点有限元的革新。作为一种矢量型有限元，棱单元因将自由度定义在棱边上而得名。实际上，早在 1957 年 `Whitney` [29] 就描述过这种类型的单元，但棱单元在电磁学中的应用及其重要性直到 20 世纪 80 年代后才逐渐被认识到。80 年代初期，Nédélec 讨论了四面体和矩形块棱单元的构造 [30] [31]。棱单元的出现解决了困扰电磁计算领域的诸多问题，将电磁场有限元分析引入一个新的时代。

本章将介绍线性棱单元在 PHG 中的具体实现，力求为用户提供友善的接口。

§3.1　棱单元介绍

棱单元在电磁场计算中扮演了重要的角色，它除了具有标准节点有限元的优点外，还有如下三点优势：

- 棱单元的计算开销与节点元大致相当。对于四面体单元，虽然棱单元离散比节点元离散得到更多的未知量 (约 6 倍)，但是，这更多的未知量被各边间较弱的联系或更稀疏的有限元矩阵所平衡。而且，比较串行程序来说，我们实现并行计算以后，内存需求已不再是主要问题。
- 对于本章开头提到的传统节点元有限元计算中碰到的三个问题，使用棱单元计算便不存在了。
- 在时谐场计算中，由于棱单元的离散势空间可计算，我们可以利用这一良好的性质设计高效的多重网格算法。

本节中，我们主要介绍实现三维线性棱单元必备的相关知识。其它关于线性棱单元的结论可以查阅 [30]。

首先，我们知道三维线性棱单元有限元的三要素 (K, P, N) 为：

- K：四面体单元。
- P：$P = \{\vec{x} \mapsto \vec{a} + \vec{b} \times \vec{x} \mid \vec{a}, \vec{b} \in R^3\}$。

- N：线性棱单元单元自由度定义为：

$$\int_e \vec{u} \cdot \vec{\tau} ds \tag{3.1}$$

其中 e 代表四面体单元的一条边，τ 表示沿边 e 的单位切向量。由此可知在每个四面体单元上有 6 个自由度。

线性棱单元的自由度只是定义在单元的边上，每条边上只有一个自由度，线性棱单元在每个单元上有 6 个基函数。

有限元计算关键的一点是基函数的选取。对于三维的四面体单元，在单元的六条边上定义线性棱单元的基函数如下：

定义 3.1.1. 设单元的第 i 条边的起点的局部编号为 m，终点编号为 n，定义该边上的基函数 W_{mn}^i 为：

$$W_{mn}^i = \lambda_m \nabla \lambda_n - \lambda_n \nabla \lambda_m \tag{3.2}$$

其中 λ 是单元的重心坐标。

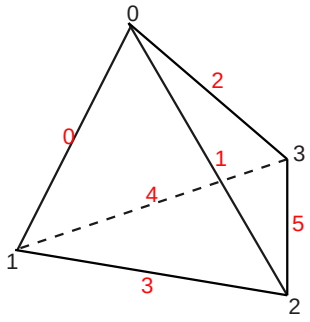

图 3.1 四面体单元边的编号

在 PHG 中，对于顶点和边之间我们有如下的约定：

```
第0条边: 两个端点为节点0和节点1;
第1条边: 两个端点为节点0和节点2;
第2条边: 两个端点为节点0和节点3;
第3条边: 两个端点为节点1和节点2;
第4条边: 两个端点为节点1和节点3;
第5条边: 两个端点为节点2和节点3;
```

如图 3.1 所示，这里所说的节点的编号为顶点在单元上的编号。

由于棱单元是矢量型有限元，其基函数 W_{mn}^i 是矢量函数（(3.2) 式中 $\nabla\lambda_n$ 是矢量)，如果将第 i 条边的起点和终点倒置，得到的 W_{mn}^i 和 W_{nm}^i 反号。因此为了保证棱单元的切向连续性，我们需要给网格中所有的边一个唯一的方向。这样避免定义在相邻单元公共边上的基函数符号不一致。我们规定边的正向为：以边的两端点中全局编号大的点作为终点，全局编号小的点作为起点。由于每个点的全局编号是唯一的，所以边的正向也是唯一的。有一点要注意是这里使用的是顶点的全局编号而不是本地编号。使用本地编号能保证在子网格内的边定向唯一，但是子网格间的内边界面上的边的定向不能保证。

基于以上考虑，在我们实现棱单元的过程中必须通过判断每条边两个顶点的全局编号的大小来对边定向，以此确定基函数的符号。

§3.2 棱单元的实现

本节将具体介绍在 PHG 中实现的棱单元自由度类型。我们将线性棱单元的自由度类型命名为 `DOF_ND1`，按照 §2.2 一节中介绍的自由度类型 `DOF_TYPE` 数据结构，将线性棱单元定义如下：

```
DOF_TYPE DOF_ND1_ = {
        DofCache,
        “Nedelec-1” ,
        DOF_ND1_points,
        DOF_DG0,
        ND1_interp, NULL, ND1_init, ND1_bas, ND1_grad,
        FALSE, FALSE,
        -1,
        NEdge, 1, -1,
        Dim,
        0, 1, 0, 0
 };
```

其中：

- `DofCache` 是 PHG 中预定义的一个宏，它与后面定义的 `-1` 都是供数值积分的 cache 机制使用，参看 §2.4 中的介绍。
- `"Nedelec-1"` 为该自由度的名称。

- DOF_ND1_points 对于线性棱单元来说没有太大的意义，我们定义一个数组 DOF_ND1_points = { 0.5, 0.5 }，将边的中点虚拟为定义自由度的节点。
- DOF_DG0 意味着该自由度梯度的类型为 Discontinuous Galerkin 元。
- ND1_interp 是粗网格到细网格的插值函数，在 §3.2.4 中介绍。
- ND1_init 是线性棱单元自由度赋值函数，在 §3.2.3 中介绍。
- ND1_bas 是计算线性棱单元基函数在单元上某点值的函数，在 §3.2.1 中介绍。
- ND1_grad 是计算线性棱单元基函数关于重心坐标的梯度在单元上某点值的函数，在 §3.2.2 中介绍。
- 两个 FALSE 表明该自由度的基函数在每个单元上是不一样的，以及该类型的自由度对象是静态定义的，使用完毕后不能释放。
- NEdge, 1, -1 表明线性棱单元在每个单元上有 NEdge 个基函数，每个基函数是一阶的，同时是间断的。NEdge 是 PHG 预定义的宏，三维时它等于 6。
- Dim 给出基函数的维数。三维时 Dim = 3。
- 0, 1, 0, 0 表示线性棱单元只在每条边上有 1 个自由度。

定义线性棱单元自由度类型最重要的是实现计算基函数、基函数导数的函数，赋值函数和插值函数。我们在接下来的四节中将分别做详细的介绍。

§3.2.1 基函数

线性棱单元基函数的计算比较简单，其函数形式为：

```
static FLOAT *
ND1_bas (GRID *g, SIMPLEX *e, int no, FLOAT *lambda)
```

该函数返回在单元 e 上重心坐标为 lambda 的点上第 no 个基函数的值，如果 no < 0 则返回所有基函数的值。函数实现过程中主要是要对边按照上一节中的约定定向。然后按照公式 3.2 计算。有一点需要注意的是计算重心座标的梯度时，我们可以通过接口函数 phgGeomGetJacobian(GRID *g, SIMPLEX *e) 获得 Jacobi 矩阵 J，由于 J 满足：

$$\begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = J \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.3}$$

因此 $\nabla\lambda_i$ 就是 Jacobi 矩阵 J 中的第 i 行的前三个分量组成的向量。

§3.2.2 基函数关于重心坐标的偏导数

ND1_grad 的接口为

```
static FLOAT *
ND1_grad (GRID * g, SIMPLEX * e, int no, FLOAT * lambda)
```

函数参数含义同 ND1_bas 函数。返回值是基函数关于重心坐标的偏导数，具体值为：

$$\begin{pmatrix} \nabla\lambda_1 & -\nabla\lambda_0 & 0 & 0 \\ \nabla\lambda_2 & 0 & -\nabla\lambda_0 & 0 \\ \nabla\lambda_3 & 0 & 0 & -\nabla\lambda_0 \\ 0 & \nabla\lambda_2 & -\nabla\lambda_1 & 0 \\ 0 & \nabla\lambda_3 & 0 & -\nabla\lambda_1 \\ 0 & 0 & \nabla\lambda_3 & -\nabla\lambda_2 \end{pmatrix} \tag{3.4}$$

梯度值的顺序按 FLOAT[nbas][dim][Dim + 1] 的方式排列。

§3.2.3 赋值函数

赋值函数接口如下：

```
void ND1_init(const DOF *dof, const SIMPLEX *e, GTYPE type, int index,
              DOF_USER_FUNC userfunc,
              DOF_USER_FUNC_LAMBDA userfunc_lambda,
              FLOAT *funcvalues, FLOAT *dofvalues)
```

该函数给指定单元内的指定自由度赋值。

- 参数 type 指定自由度类别，VERTEX 表示顶点上的自由度，EDGE 表示边上的自由度，FACE 表示面上的自由度，ELEMENT 表示单元体内的自由度。对于线性棱单元而言只需要处理 type = EDGE 的情况。
- 参数 index 给出顶点、边或面在单元中的编号。对于棱单元意味着边的编号(边在单元上的编号)。
- 参数 userfunc、user_func_lambda 和 funcvalues 分别为两个函数指针和一个数组指针，它们给出计算函数值的函数指针或包含函数值的缓冲区地址，三个指针中必须有且仅有一个为非空指针。userfunc 指向一个关于 x, y, z 的函数，其接口类型为 DOF_USER_FUNC，具体如下：

```
void userfunc(FLOAT x, FLOAT y, FLOAT z, FLOAT *values)
```

而 userfunc_lambda 则指向一个关于重心坐标的函数，其接口类型为 DOF_USER_FUNC_LAMBDA，具体如下：

```
void userfunc_lambda(DOF *dof, simplex *e, FLOAT lambda[],
                     FLOAT *values)
```

两个函数均需将指定坐标处的函数值，共计 dof->dim×3 个值，填写在 values 所指向的缓冲区中。

当 funcvalues 为非空指针时，它指向存有边中点处的 dof->dim × 3 个函数值的缓冲区，ND1_init 根据这些函数值计算自由度的值。

- 参数 dofvalues 指向存储自由度值的缓冲区，它的长度为 dof->dim。ND1_init 需要计算指定边的自由度值，并将结果放在 dofvalues 指向的缓冲区中返回给调用程序，该缓冲区由调用的程序提供，长度为 dof->dim。

根据棱单元上的自由度计算公式（式 (3.1)），我们给出线性棱单元赋值函数的具体实现：

```
procedure ND1_init(dof,e,type, index, userfunc, funcvalues, values)
do
 给边定向，获取边起点和终点的座标 P0(x0,y0,z0) 和 P1(x1,y1,z1)
 if (funcvalues == NULL); then
   利用 userfunc 或 userfunc_lambda 计算式 (3.1) 中 u 在边中点的值
   结果保存在向量数组 v[dof->dim];
 fi
 l = P1 - P0
 values[0 ··· dof->dim-1]= v[0 ··· dof->dim-1] ·l
done
```

由于线性棱单元的基函数是一阶的，因此我们这里计算式 (3.1) 时使用的是一阶精度的数值积分。

§3.2.4 插值函数

自由度类型中定义的插值函数有两个，一个是细网格到粗网格的插值函数，另外一个是粗网格到细网格的插值函数。其中细网格到粗网格的插值函数用于网

表 3.1 粗网格到细网格插值函数中父亲单元 DOF 数据

parent_data 项 (dim = dof->dim)	数据块内容
(parent_data[i])[np_vert][dim]	顶点 i 上的自由度值, $i = 0, 1, 2, 3$
(parent_data[$4 + i$])[np_edge][dim]	边 i 上的自由度值, $i = 0, 1, 2, 3, 4, 5$
(parent_data[$10 + i$])[np_face][dim]	面 i 上的自由度值, $i = 0, 1, 2, 3, 4$
(parent_data[14])[np_elem][dim]	单元体中的自由度值

格粗化，粗网格到细网格的插值函数用于网格细化。本文中的算例主要涉及网格细化操作，因此我们主要介绍粗网格到细网格的插值函数。其接口为：

```
void ND1_interp(DOF *dof, SIMPLEX *e, FLOAT **parent_data,
                FLOAT **child_data)
```

其中 dof 为自由度对象，e 为父亲单元。数组 parent_data 中包含 15 个指针，指向父亲单元中的自由度数据，这里每个顶点、边、面和体中的自由度数据都是连续存放的。具体地，这些指针的含义在表 3.1 中给出，表中 dof->dim 为自由度对象的维数（参看 §2.2）。

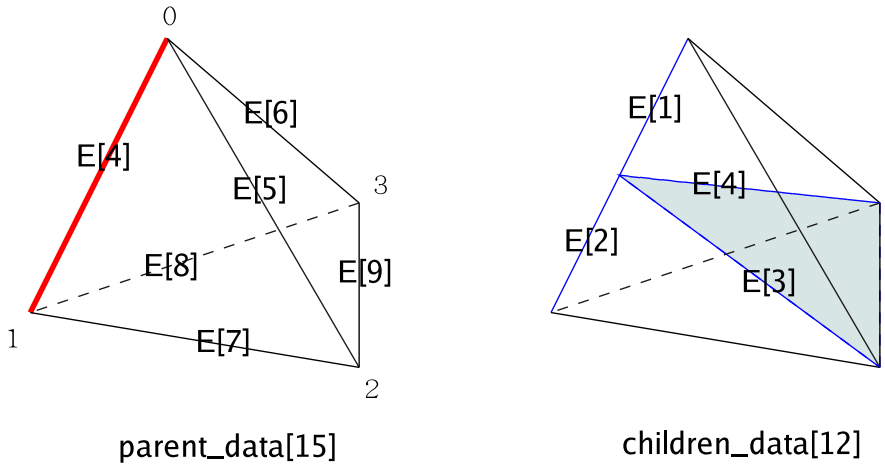

图 3.2 插值函数中父、子单元边自由度数据的编号

一个单元细化后产生一个新顶点、四条新边、5 个新面和两个新单元，相应地，数组 child_data 中包含 12 个指针，分别指向这些数据块，这些指针的含义在表 3.2 中给出。

图 3.2 给出了插值函数中父、子单元边的自由度数据编号示意。

表 3.2 粗网格到细网格插值函数中子单元 DOF 数据

child_data 项 dim=dof->dim	数据块内容
(child_data[0])[np_vert][dim]	新顶点上的自由度值
(child_data[1])[np_edge][dim]	新顶点和老顶点 0 构成的边上的自由度值
(child_data[2])[np_edge][dim]	新顶点和老顶点 1 构成的边上的自由度值
(child_data[3])[np_edge][dim]	新顶点和老顶点 2 构成的边上的自由度值
(child_data[4])[np_edge][dim]	新顶点和老顶点 3 构成的边上的自由度值
(child_data[5])[np_face][dim]	新顶点和老顶点 0, 3 构成的面上的自由度值
(child_data[6])[np_face][dim]	新顶点和老顶点 1, 3 构成的面上的自由度值
(child_data[7])[np_face][dim]	新顶点和老顶点 0, 2 构成的面上的自由度值
(child_data[8])[np_face][dim]	新顶点和老顶点 1, 2 构成的面上的自由度值
(child_data[9])[np_face][dim]	新顶点和老顶点 2, 3 构成的面上的自由度值
(child_data[10])[np_elem][dim]	包含老顶点 0 的新单元体上的自由度值
(child_data[11])[np_elem][dim]	包含老顶点 1 的新单元体上的自由度值

由于棱单元的自由度全在边上，因此在实现插值函数时我们只需要关注：parent_data[4,...,9] 和 child_data[1,...,4]。

在自适应网格细化的时候，一个单元被细化产生两个子单元，会产生一个新的顶点和 4 条新边，如图 3.3 所示。PHG 中约定，对单元编号为 0 和 1 的节点之间的边进行细化，记产生的新顶点为 nv，四条新边为：$\overrightarrow{\mathrm{ne}_i} = (i, \mathrm{nv}),\ i = 0, 1, 2, 3$（$\mathrm{ne}_i$ 的两个端点为 i 和 nv）。

对于棱单元而言，一次细化产生了 4 个新的自由度。记父亲单元的 6 条边为 $\overrightarrow{e_0}$ (0–1)、$\overrightarrow{e_1}$ (0–2)、$\overrightarrow{e_2}$ (0–3)、$\overrightarrow{e_3}$ (1–2)、$\overrightarrow{e_4}$ (1–3)、$\overrightarrow{e_5}$ (2–3)。将父亲单元自由度的值进行简单插值，算出新自由度的值（即为 $child\_data[1, \cdots, 4]$。具体算法为：

```
procedure ND1_interp(dof, e, parent_data, child_data)
  对细网格上的边 ne_i ; i = 0,...,3 和粗网格上的边 e_j ; j = 0,...,5 定向
  child_data[1]=
    0.5 × sign(ne_0, e_0) × parent_data[4]
  child_data[2]=
    0.5 × sign(ne_1, e_0) × parent_data[4]
  child_data[3]=
    0.5 × sign(ne_2, e_1) × parent_data[5] + 0.5 × sign(ne_2, e_3) × parent_data[7]
```

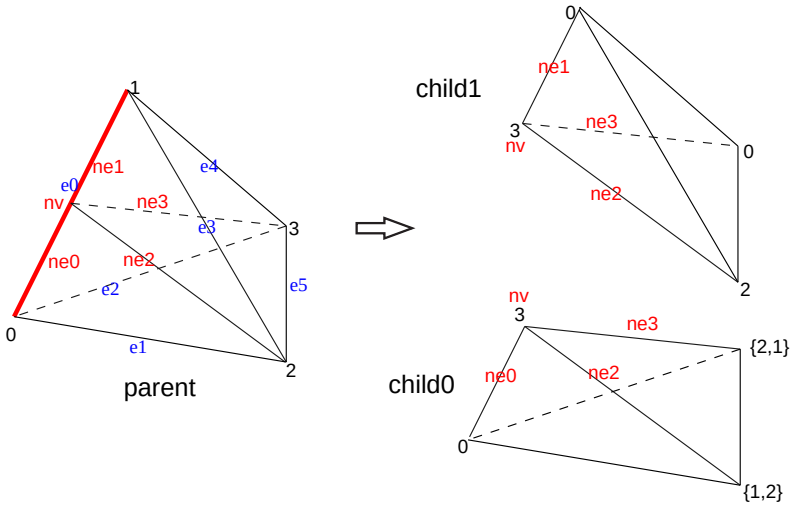

图 3.3 父亲单元与子单元编号关系

```
child_data[4]=
    0.5 × sign(ne2, e2) × parent_data[6] + 0.5 × sign(ne2, e4) × parent_data[8]
```

其中，

$$\mathrm{sign}(\overrightarrow{\mathrm{ne}_i}, \overrightarrow{e_j}) = \begin{cases} 1, & \text{边 } \overrightarrow{\mathrm{ne}_i} \text{ 与边 } \overrightarrow{e_j} \text{ 方向相同} \\ -1, & \text{边 } \overrightarrow{\mathrm{ne}_i} \text{ 与边 } \overrightarrow{e_j} \text{ 方向相反} \end{cases}$$

第四章　时谐场问题数值模拟

在第一章引言中，已经简单介绍过 Maxwell 方程组。本章针对当 Maxwell 方程组中的场量是单频的谐振函数时的时谐场数学模型，使用在 第三章 中介绍的线性棱单元进行并行自适应数值实验。我们将给出三个算例，以此来验证 PHG 并行自适应框架的正确性和性能，以及使用本文实现的线性棱单元进行自适应有限元计算的有效性。为了便于比较，本文选用了 [45] 中的三个算例，[45] 中给出了它们在 SGI Origin 3800 上的串行数值实验的结果。

§4.1　齐次 Dirichlet 边界条件的时谐场问题

在 §1.3 节中，我们可以使用复相位因子表示法 [41]，将（1.1）式和（1.2）式以及（1.5）式写成简单的形式：

$$\begin{aligned} \operatorname{curl}\vec{E} + i\omega\vec{B} &= 0 && (4.1)\\ \operatorname{curl}\vec{H} - i\omega\vec{D} &= \vec{J} && (4.2)\\ \nabla\cdot\vec{J} &= -i\omega\rho && (4.3) \end{aligned}$$

利用本构关系式 (1.6)–(1.8 式)，从式 (4.1) 和式 (4.2) 中消去 $\vec{H}$，可以得到 $\vec{E}$ 的微分方程：

$$\operatorname{curl}(\frac{1}{\mu}\operatorname{curl}\vec{E}) + (i\sigma\omega - \omega^2\varepsilon)\vec{E} = -i\omega\vec{J}_s \tag{4.4}$$

其中 $\vec{J}_s$ 是外加电流或源电源。

定义 $k^2 = \omega^2\varepsilon - i\sigma\omega$，$J = -i\omega\vec{J}_s$，则式 (4.4) 变为：

$$\operatorname{curl}(\frac{1}{\mu}\operatorname{curl}\vec{E}) - k^2\vec{E} = \vec{J}, \quad \nabla\cdot\vec{J} = 0, \qquad \text{in } \Omega \tag{4.5}$$

我们假定求解区域 $\Omega \subset R^3$ 为单联通的多面体区域，且是 Lipschitz 边界。我们取齐次 Dirichlet 边界条件：

$$\vec{E}\times\vec{n} = 0, \qquad \text{on } \partial\Omega \tag{4.6}$$

其中 $\vec{n}$ 为边界面上的外法向量。和式 (4.5) 与式 (4.6) 等价的变分问题为：
求 $\vec{u} \in H_0(\operatorname{curl},\Omega)$，使得：

$$a(\vec{u},\vec{v}) = (\vec{J},\vec{v}), \qquad \forall\vec{v} \in H_0(\operatorname{curl};\Omega) \tag{4.7}$$

其中：

$$\begin{aligned} a(\vec{u},\vec{v}) &= (\frac{1}{\mu}\operatorname{curl}\vec{u},\operatorname{curl}\vec{v}) - (k^2\vec{u},\vec{v}) && (4.8)\\ H_0(\operatorname{curl};\Omega) &:= \{\vec{u}\in L^2(\Omega);\ \operatorname{curl}\vec{u}\in L^2(\Omega);\ \vec{u}\times\vec{n} = 0 \text{ on } \partial\Omega\} && (4.9) \end{aligned}$$

本文选用线性棱单元来进行时谐场问题 (4.7) 的有限元离散。设 $\mathcal{T}_h$ 为求解区域 Ω 的拟一致、正则的四面体剖分，对任何单元 t，定义单元上的基函数空间为 $\mathcal{ND}_1(t) := \{x \mapsto a + b \times x, a, b \in R^3\}$，其全局有限元空间为 $\mathcal{ND}_1(\mathcal{T}_h) := \{\eta_h \in H(\mathrm{curl};\Omega); \eta_h|t \in \mathcal{ND}_1(t), \forall t \in \mathcal{T}_h\}$。记 $\mathcal{ND}_{1,0}(\mathcal{T}_h)$ 为满足齐次 Dirichlet 边界条件的子空间。则问题 (4.7) 相应的有限元离散为：

给定谐调有限元空间 $\mathcal{ND}_{1,0}(\mathcal{T}_h) \subset H_0(\mathrm{curl};\Omega)$，求 $\vec{u_h} \in \mathcal{ND}_{1,0}(\mathcal{T}_h)$，使得 $\forall \vec{v_h} \in \mathcal{ND}_{1,0}(\mathcal{T}_h)$ 有

$$a(\vec{u_h}, \vec{v_h}) = (\frac{1}{\mu}\,\mathrm{curl}\,\vec{u}_h, \mathrm{curl}\,\vec{v}_h)_{0,\Omega} - (k^2\vec{u}_h, \vec{v}_h)_{0,\Omega} = (\vec{J}, \vec{v_h}) \tag{4.10}$$

时谐场问题的 Dirichlet 边界条件是 $\vec{E}$ 与边界面单位法向量 $\vec{n}$ 的外积。如果 $\vec{E} \times \vec{n} = g$，那么对于线性棱单元，其边界上自由度的值为

$$\int_e E \cdot \vec{l}\,\mathrm{d}s = \int_e g \cdot (\vec{l} \times \vec{n})\,\mathrm{d}s \tag{4.11}$$

其中 $\vec{l}$ 表示位于边界面上的边 e 的切向。

§4.2 合成线性系统

我们定义一个棱单元自由度类型的对象 `u_h` 用来存储问题（4.7）的有限元解。遍历每个单元，在每个单元上生成单元刚度矩阵。通过调用 PHG 中解法器的相关接口函数，我们可以非常方便地合成总刚度矩阵以及右端项。然后求解该线性系统获得式 (4.7) 的有限元解。

假设 W_i^e 代表单元 e 上的第 i 个基函数。根据式 (4.10)，进行有限元离散后得到的单元刚度矩阵包含以下两个积分：

$$E_{ij}^e = \iiint_e (\mathrm{curl}\,W_i^e) \cdot (\mathrm{curl}\,W_j^e)\,\mathrm{d}V \tag{4.12}$$

$$F_{ij}^e = \iiint_e W_i^e \cdot W_j^e\,\mathrm{d}V \tag{4.13}$$

而线性系统的右端项需要计算积分：

$$\hat{B}_i^e = \iiint_e J \cdot W_i^e\,\mathrm{d}V \tag{4.14}$$

对网格中单元的遍历通过宏 `ForAllElements` 来完成。

假设线性方程组的系数矩阵为 $A(I,J)$，右端向量为 $B(I)$，$I, J = 0, \ldots, N-1$，N 为未知量个数，则组装线性系统的过程如下：

```
NBas = u_h->type->nbas;              /* 每个单元中的基函数个数 */
ForAllElements(g, e) {               /* 对单元进行遍历 */
    for (i = 0; i < NBas; i++) {
        I = phgSolverMapE2L(solver, 0, e, i);
        type = phgDofGetElementBoundaryType(u_h, e, i);
        if (type & DIRICHLET) {
            continue;
        }
        for (j = 0; j < NBas; j++) {
            J = phgSolverMapE2L(solver, 0, e, j);
            计算 E_ij - k^2 F_{i,j} 并累加到 A(I,J);
        }
        计算 B̂_i 并累加到 B(I);
    }
    /* 对单元的每个面循环，处理边界条件 */
    For (i = 0; i < NFace; i++) {
       if (e->bound_type[i] & DIRICHLET) {
            /*处理边界条件*/
       }
    }
}
```

在上述计算中，函数 phgSolverMapE2L(solver, 0, e, i) 计算单元 e 中对应 solver 的第 0 个 DOF 对象（即 u_h）中的第 i 个自由度在方程组中的本地未知量编号，phgDofGetElementBoundaryType 返回该自由度的边界条件类型。E_{ij} 的计算调用函数 phgQuadCurlBasDotCurlBas 完成。F_{ij} 的计算调用函数phgQuadBasDotBas 完成。$\hat{B}_i$ 的计算调用函数 phgQuadDofDotBas 或者 phgQuadFuncDotBas 完成。这里有必要说明的是如果将问题的右端函数 J 也在网格上离散，视为自由度，可以使用 phgQuadDofDotBas 计算 $\hat{B}_i$，只不过这样会引入有限元解到问题解的插值误差，而用 phgQuadFuncDotBas 将不会有这个问题，但当右端项函数比较复杂时，前者的计算量比后者要小。从实际计算过程中发现，对于大部分例子当网格很密以后，两种做法对结果的影响不大。但是对于某些情况如本文中的算例一（§4.4.2），第一种做法的计算结果会出现问题。

由于棱单元边界条件的处理需要用到边界面的外法向，因此我们通过对单元的面遍历，利用（4.11）式处理边界条件。

§4.3 自适应网格细化

自适应有限元计算最关键的部分是后验误差估计。只有正确计算了后验误差，然后根据适当的网格细化策略，才能有效地进行自适应计算。在本文的例子中，我们选取以下的后验误差估计子：

$$\eta_T^2 = h_T^2(\|f + k^2 u_h - \mathrm{curl}\,\frac{1}{\mu}\,\mathrm{curl}\,u_h\|_{0,T}^2 + \|\nabla\cdot f + k^2 u_h\|_{0,T}^2) \tag{4.15}$$

$$\eta_F^2 = h_F(\|[\frac{1}{\mu}(\mathrm{curl}\,u_h)\times\vec{n}]\|_{0,F}^2 + \|[(f + k^2 u_h)\cdot\vec{n}]\|_{0,F}^2 \tag{4.16}$$

其中 $[\cdot]$ 表示面上的跳量，h_T 为单元 T 的直径，h_F 为面 F 的直径。

关于该后验误差估计子在 [5] [45] 中有详细的证明，这里只给出结论。

定理 4.1. *设 u 和 u_h 分别是问题 (4.7) 和 (4.10) 的解，$\mathcal{T}_h$ 和 $\mathcal{F}_h$ 分别代表网格上所有单元和所有面的集合，则*

$$||u - u_h||_{H(curl;\Omega)} \le C(\sum_{T\in\mathcal{T}_h}\eta_T^2 + \sum_{T\in\mathcal{F}_h}\eta_F^2)^{\frac{1}{2}} \tag{4.17}$$

C 依赖于 μ^{-1}，k^2 和单元的最小角。

本文中的算例使用最大误差策略（见 §1.1）作为细化策略，选取 $\gamma = 0.5$。

§4.4 数值算例

所有算例都是在“科学与工程计算国家重点实验室”的联想深腾 1800 (LSSC-II) 和 SGI Origin3800 超级计算机上完成的。

LSSC-II 有 256 个计算结点、每个计算结点拥有两个 Intel 2GHz Xeon 处理器和 1GB 内存，浮点运算总性能为 2Tflops。计算结点上同时配备了快速以态网和 Myrinet 2000，我们使用 Myrinet 2000 作为 MPI 通信网络。

SGI Origin3800 包含 64 个处理器，处理器主频为 600MHz，采用 MIPS R14000 CPU，总内存为 64GB。

§4.4.1 算例一

计算区域为 $\Omega = (0,3)^3 \setminus (0.5,1)^3$，材料系数为 $\mu = 1$，$k^2 = -1$，真解为：

$$\begin{pmatrix} xyz(x-1)(y-1)(z-1)(x-0.5)(y-0.5)(z-0.5) \\ \sin(2\pi x)\sin(2\pi y)\sin(2\pi z) \\ (1-e^x)(e-e^x)(e-e^{2x})(1-e^y)(e-e^y)(e-e^{2y})(1-e^z)(e-e^z)(e-e^{2z}) \end{pmatrix}$$

我们籍由这个算例来检验程序的正确性、鲁棒性和并行效率，同时也验证棱单元函数的逼近性质、自适应后验误差估计的有效性。

算例一中，线性系统求解使用 PETSc 提供的 MINRES 迭代法，使用 Jacobi 预条件子，迭代终止的残量阀值为 1.0E-12。

从表 4.1 中，我们看出当计算规模都为 688128 个单元时，在 LSSC-II 和 Origin3800 上，生成线性系统的时间和求解线性系统的时间都是随着处理器个数的增长而线性下降的，如图 4.1 所示。程序表现出很好的并行效率和可移植性。

表 4.1 o3800/LSSC-II 上并行效率分析 (算例一，一致网格细化，单元数 688128)

处理器个数	生成线性系统时间（秒）	求解线性系统时间
1	89.5516/88.5221	919.0556/153.9043
2	46.4479/45.1204	495.7075/81.0481
4	23.4138/22.7706	281.0478/41.6107
8	15.6520/11.3673	184.7517/21.0269
16	8.5290/5.6733	60.3238/10.8781
32	(LSSC-II) 2.8317	(LSSC-II) 5.3838
64	(LSSC-II) 1.4079	(LSSC-II) 3.2265

对一致网格细化，我们检验了不同的预条件子对线性系统求解时间的影响。表 4.2 显示使用 SOR 预条件子所需的迭代步数少于其它预条件子；但是使用 Jacobi 预条件子的求解时间优于其它预条件子。原因在于 SOR 预条件的处理所花的时间明显比 Jacobi 预条件所花的时间要长。

表 4.3 中给出了 SGI Origin 3800 上使用一个处理器的计算时间与 [45] 中基于 ALBERTA 及棱单元计算本算例的计算时间的比较，包括生成线性系统时间（秒）、求解时间（秒）及网格细化时间（秒）。两个计算中求解线性系统都使用了 GMRES 方法及 Jacobi 预条件子。由于 [45] 中棱单元处理和组装线性系统的代码是在 ALBERTA 的基础上由该文作者自行编制的，其性能并不完全代表

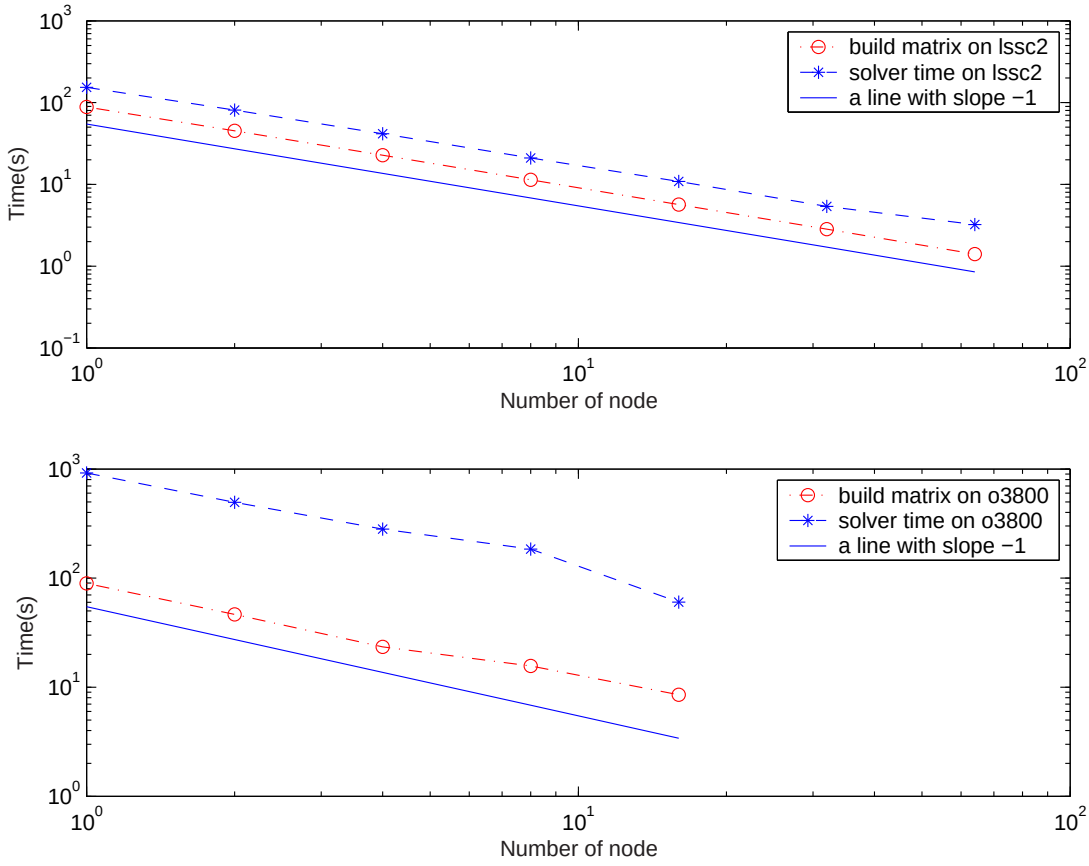

图 4.1 并行效率图（算例一）

表 4.2 不同预条件子的求解时间和迭代次数（算例一，一致网格细化，LSSC-II 8 结点）

单元数	使用不同预条件子的求解时间（秒）/迭代次数（次）				
	None	Jacobi	BJacobi	SOR	Add. Schwarz
168	0.01/124	0.01/101	0.01/48	0.01/36	0.03/48
1344	0.10/359	0.07/236	0.11/247	0.12/206	0.13/161
10752	0.46/781	0.28/452	0.40/446	0.35/356	0.51/408
86016	6.16/1289	3.54/716	5.91/776	4.86/596	6.62/730
688128	91.90/2091	48.69/1080	82.41/1196	68.06/893	97.60/1251
5505024	1286.26/3536	531.35/1439	952.42/1646	729.62/1196	1215.61/1916

ALBERTA 的性能。从表 4.3 中我们可以发现我们的程序生成线性系统所需时间是 [45] 中所需时间的 60% 左右。而在求解时间的比较中，PHG 使用的 PETSc 求解器亦明显优于 [45] 中使用的 ALBERTA 求解器的性能。PHG 的网格细化时间要大大长于 ALBERTA 的网格细化时间，这是因为 PHG 的网格细化时间统计中包含了有限元解和几何数据 (Jacobian、法向量、单元直径等) 的插值计算以及右端项函数投影到细网格有限元空间的计算。由于本算例中使用的右端项函数非常复杂，因此实际上 PHG 所统计的网格细化时间绝大部分用于右端项的计算。

表 4.3 PHG 与 ALBERTA 效率比较

自由度数	生成线性系统时间（秒）		求解时间（秒）		网格细化时间（秒）	
/单元数	ALBERTA	PHG	ALBERTA	PHG	ALBERTA	PHG
281/168	0.0319	0.0209	0.1469	0.0141	0.0015	0.0159
1924/1344	0.2661	0.1639	8.1545	0.2047	0.0139	0.1131
14168/10752	2.2598	1.3344	282.3768	42.6511s	0.1206	1.8333
108592/86016	19.0378	10.9480	10814.2905	464.2752	1.1172	7.6673

表 4.4 给出随着网格的一致细化，有限元解与真解之间的 $H(\mathrm{curl};\Omega)$ 误差以及有限元解的后验误差估计子的下降情况。从表中数据可以看出，随着网格的一致细化，棱单元有限元的插值误差 $\|u-u_h\|_{H(\mathrm{curl},\Omega)}$ 是 $O(h)$ 阶的，其中 h 表示网格步长。

表 4.5 给出了自适应细化下的数值结果。图 4.3 是自适应后验误差 η 与自由度数 N 之间的 log–log 图。由于自适应的“最优”体现在：$\eta_i/\eta_0 \sim (N_i/N_0)^{(-1/3)}$，所以理想的状况应该是图中的实验数据连成的折线“平行”于斜率为 $-1/3$ 的基准线。由于解在区域内很光滑，所以不管全局细化还是自适应细化的结果都很理想。但是如果解具有很强的奇性，则自适应的优势就体现出来了，如算例二和算例三的图 4.8、图 4.14 所示。从图 4.2 可以看出，棱单元有限元的插值误差 $\|u-u_h\|_{H(\mathrm{curl},\Omega)}$ 与网格单元数之间也存在类似的关系。从 $\|u-u_h\|_{H(\mathrm{curl},\Omega)}$ 的下降趋势也能说明 PHG 中实现的棱单元的正确性。

图 4.4 和图 4.5 分别是真解和有限元解在 $z=0.1$ 平面上的模，图 4.6 和图 4.7 是两者的矢量图表示。从图中可以看出，棱单元解基本上抓住了真解的特性。

从表 4.4 和表 4.5 可以看出，求解时间和自由度总数也基本上呈线性关系。但是自适应网格细化后生成的线性系统条件数比使用一致细化生成的线性系统的条件数要差，如果只是使用简单的预条件子，自适应方法求解线性系统的时间也相对长一些。例如一致网格细化达到 88080384 个单元数时，求解时间为 5074.2140，

迭代步数为 8972 步；而自适应网格细化达到 64767606 个单元数时，求解时间为 5394.3798 秒，迭代步数为 15382 步。这种现象在算例三和算例四中更明显，因此有必要研究求解 Maxwell 方程有限元离散所形成的线性系统的更好的算法和预条件子，如多重网格预条件子。

最后要指出的是，在数值实验中，当自由度的个数达到千万阶以后 (一致网格细化时自由度数达到 103351040) 程序的各个模块仍然表现正常，这说明程序具有很好的鲁棒性。

综上所述，实验数据与理论分析的结果是吻合的，说明本文实现的棱单元模块和自适应模块是可靠的，具有很好的并行可扩展性。

表 4.4 一致细化的情况，迭代终止残量为 1.0E-12（算例一，128 结点）

网格层号	自由度数/单元数	$\|u-u_h\|_{H(\mathrm{curl},\Omega)}$	后验误差估计值	求解时间（秒）/迭代步数（次）
1	548/336*(3)	1.124e+00	2.120e+00	0.0521/129
2	3736/2688*(26)	7.831e-01	1.091e+00	0.2298/378
3	27440/21504*(128)	4.046e-01	5.661e-01	0.7798/804
4	210016/172032	2.045e-01	2.870e-01	2.2982/1601
5	1642688/1376256	1.028e-01	1.443e-01	23.0335/3009
6	12992896/11010048	5.153e-02	7.234e-02	373.5315/5174
7	103351040/88080384	2.580e-02	3.621e-02	5074.2140/8972

* () 内注明实际使用的处理器数

§4.4.2 算例二

取 $\Omega=(-1,1)^3$，$\sum=\{y=0,(x,z)\in(-\frac{1}{2}\times\frac{1}{2})^2\}$，$\mu=1$，$k^2=1$，$\vec{J}=(1,1,1)$，在 $\sum$ 平面上取 `Dirichlet` 边界条件，从而使得问题具有很强的奇性。

算例二和算例三主要用于考察在计算带有奇性的问题时，棱单元模块和自适应模块的有效性。求解器选用 PETSc 的 LGMRES ("loose" GMRES) [3] 迭代法，迭代终止的残量阀值为 1.0E-10。

表 4.6 中列出了一致网格细化在不同网格层的单元数，后验误差的值以及求解时间和单结点内存使用量。表 4.7 列出了自适应网格的相关数据。我们使用 128 个计算结点，一致细化时计算规模达到了 100663296 个单元，118080000 自由度，单结点的内存使用量为 425.3MB（占结点总内存的 43% 左右）说明我们的程序具有很好的鲁棒性和可扩展性。有一点需要说明的是，内存使用量的统计没有在

表 4.5 自适应细化的情况，迭代终止残量为 1.0E-12（算例一，128 结点）

网格层号	自由度数/单元数	$\|u-u_h\|_{H(\mathrm{curl},\Omega)}$	后验误差估计值	求解时间（秒）/迭代步数（次）
1	548/336 *(3)	1.124e+00	2.120e+00	0.1061/129
2	667/438 *(4)	1.263e+00	2.004e+00	0.0267/183
3	1087/768 *(7)	1.022e+00	1.661e+00	0.1713/228
4	1974/1406 *(14)	9.166e-01	1.273e+00	0.2742/365
5	3655/2698 *(26)	8.076e-01	1.074e+00	0.2254/480
6	6403/4878 *(48)	6.869e-01	9.186e-01	0.7794/561
7	11881/9196 *(91)	5.204e-01	7.308e-01	0.6606/819
8	19957/15660 *(128)	4.236e-01	6.004e-01	1.6274/952
19	13890446/11705316	5.031e-02	6.808e-02	574.4811/10140
20	24153631/20357052	4.366e-02	5.816e-02	1329.0700/12805
21	47995029/40140398	3.160e-02	4.548e-02	3088.0109/14257
22	77126046/64767606	2.645e-02	3.788e-02	5394.3798/15382

* () 内注明实际使用的处理器数

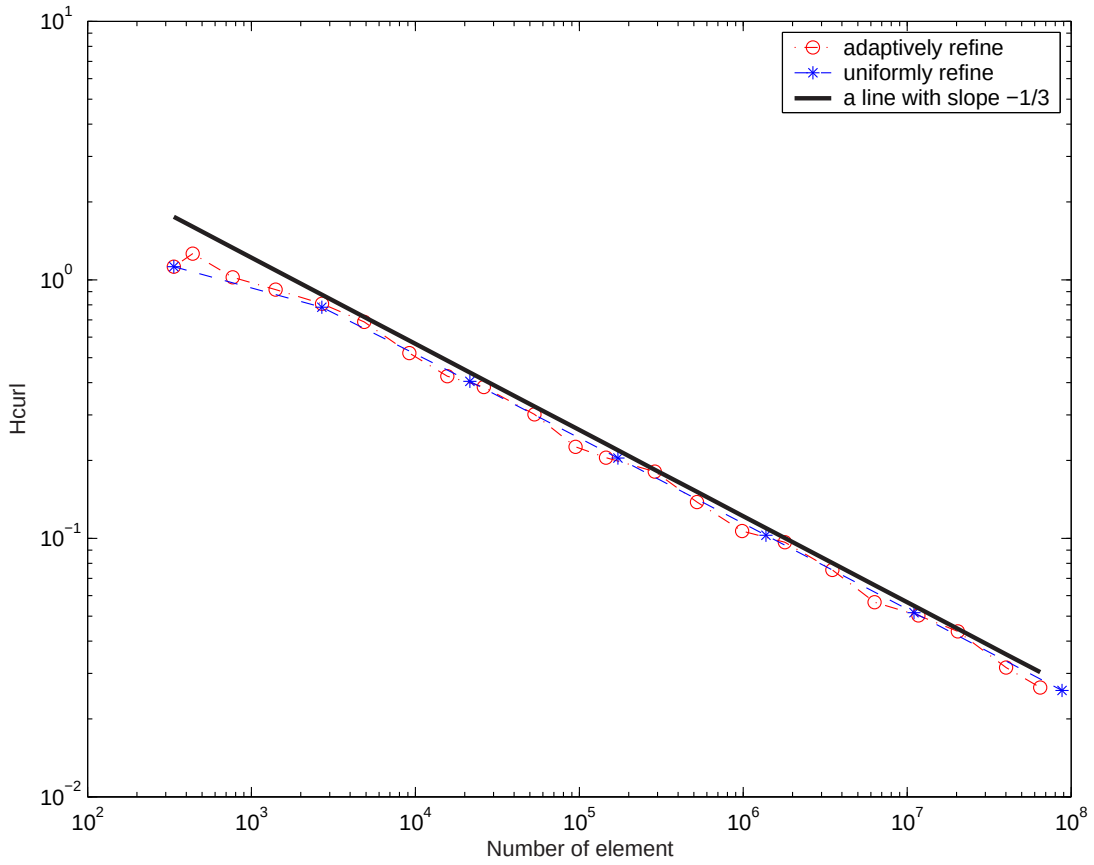

图 4.2 $\|u-u_h\|_{H(\mathrm{curl})}$ 在一致网格细化和自适应网格细化中的减小（算例一，LSSC-II 128 结点）

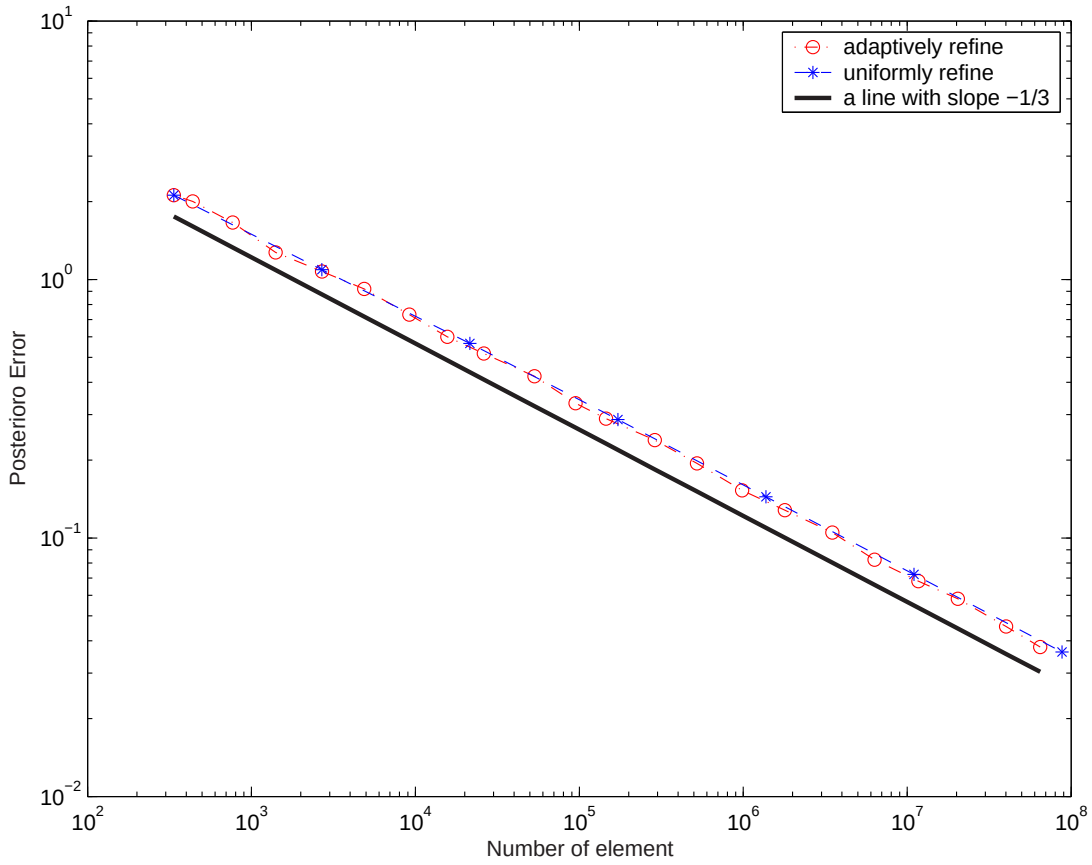

图 4.3 后验误差估计子在一致网格细化和自适应网格细化中的减小（算例一，LSSC-II 128 结点）

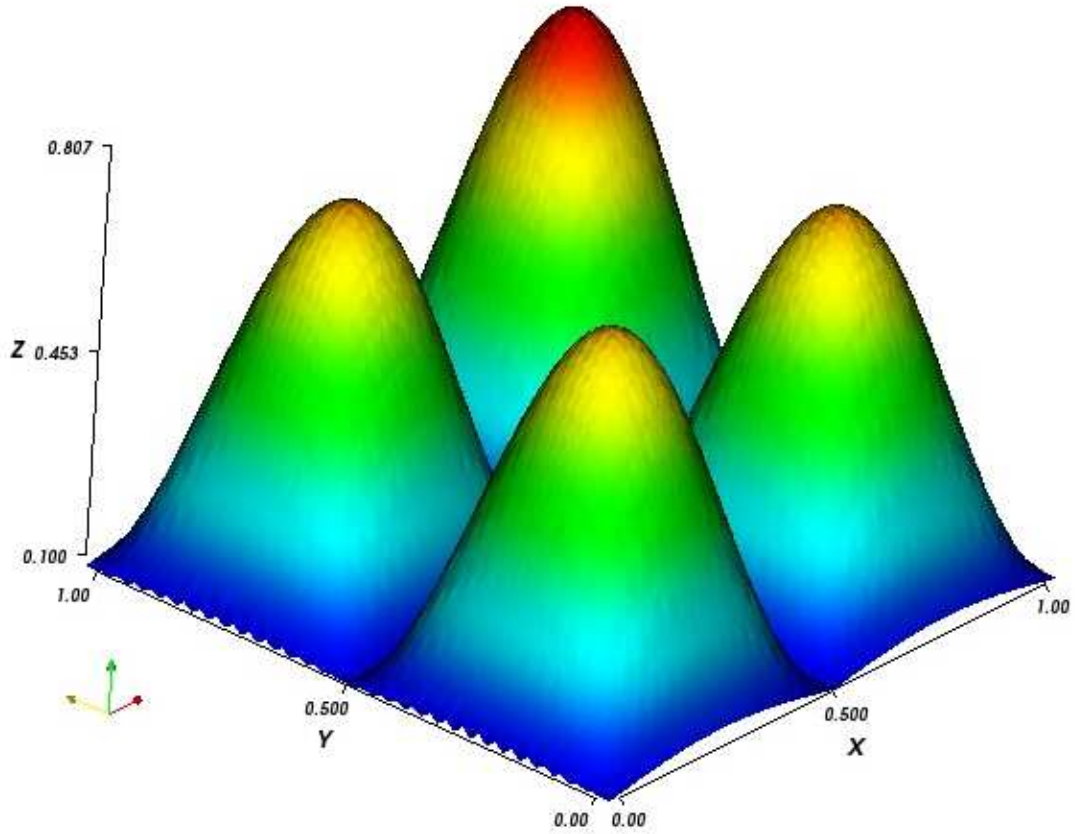

图 4.4 z=0.1 平面上真解向量的模（算例一）

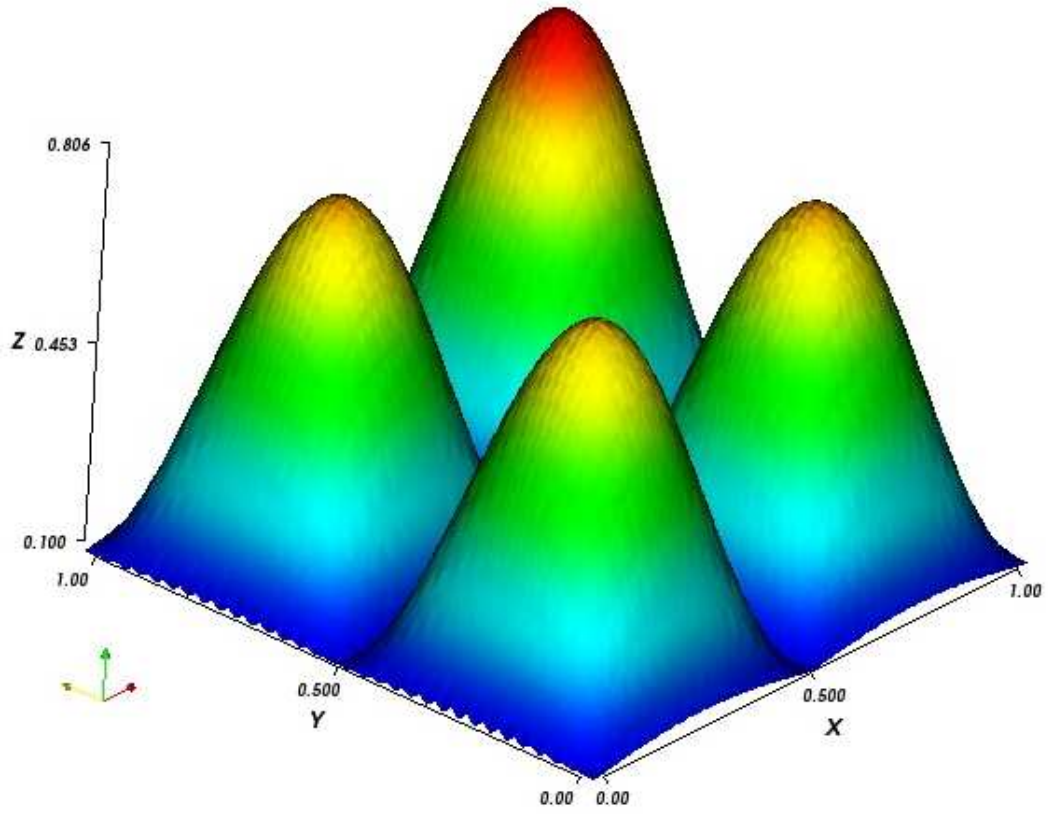

图 4.5 z=0.1 平面上有限元解向量的模（算例一）

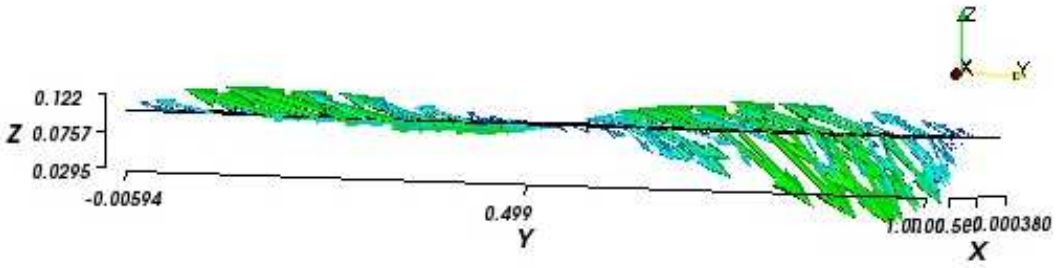

图 4.6 z=0.1 平面上真解向量（算例一）

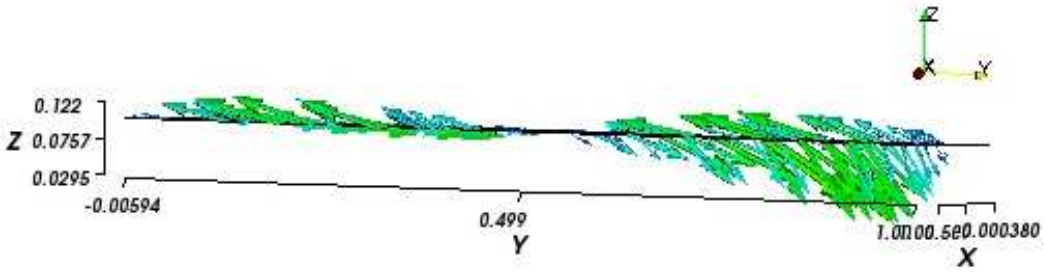

图 4.7 z=0.1 平面上有限元解向量（算例一）

PETSc 内部采样，统计结果是除去调用 PETSc 的解法器之外的最大值。根据实验发现，PETSc 的解法器还要多用约 50%-100% 的内存量。

从图 4.9 和图 4.10 可以看出，在 $\sum$ 平面的自适应网格非常好的抓住了问题的奇性，同时图 4.8 显示，自适应网格细化中的后验误差估计基本得到了最优的收敛阶，而全局细化做不到这一点。这说明我们的自适应算法是成功的。程序的棱单元模块和自适应模块是有效的。

表 4.6 一致细化的情况，迭代终止残量为 1.0E-10（算例二，128 结点）

网格层号	自由度数/单元数	后验误差估计值	求解时间（秒）/迭代步数（次）	单结点内存使用量 (MB)
1	612/384*(3)	9.329e-01	0.0394/236	32.79
2	4224/3072*(30)	4.379e-01	0.6187/1080	33.64
3	31200/24576*(128)	2.350e-01	5.4443/6146	33.15
4	239424/196608	1.356e-01	10.1277/7256	34.18
5	1875072/1572864	8.344e-02	184.2217/22402	42.54
6	14840064/12582912	5.415e-02	2721.5690/42796	113.1
7	118080000/100663296	3.647e-02	93260.7354/88679	425.3

* () 内注明实际使用的处理器数

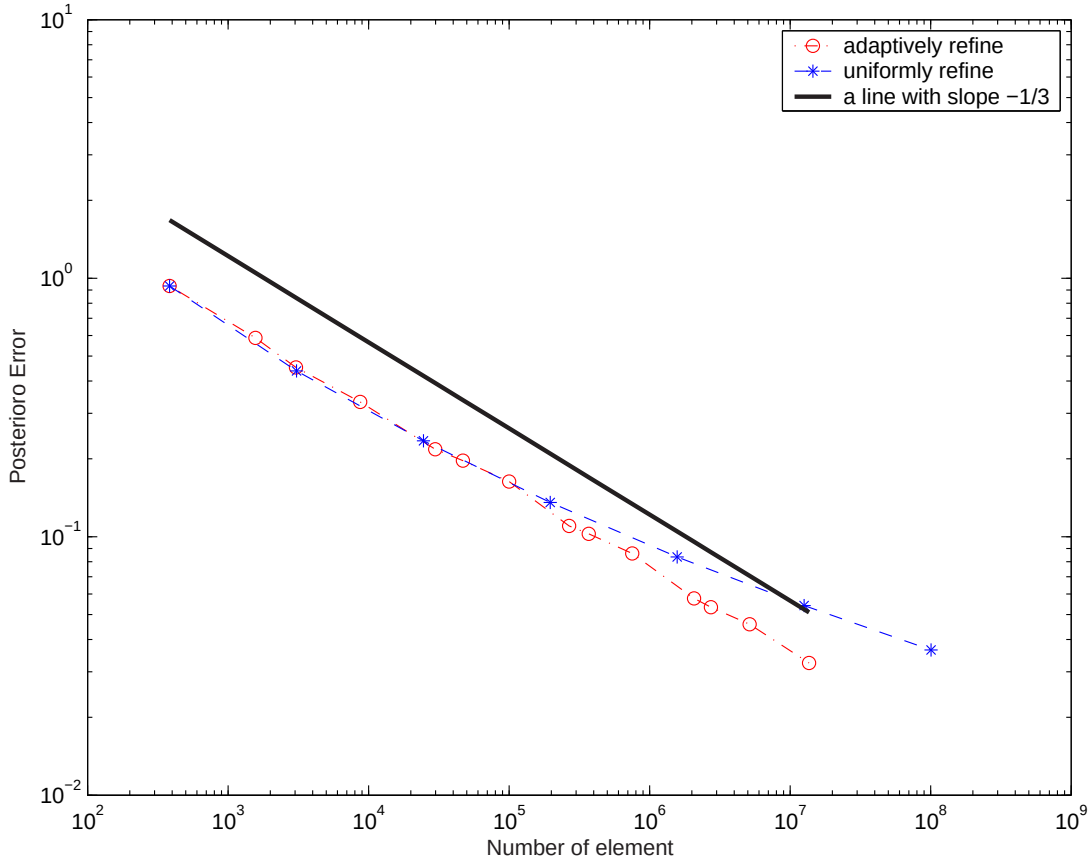

图 4.8 后验误差估计子在一致网格细化和自适应网格中的减小（算例二，LSSC-II 128 结点）

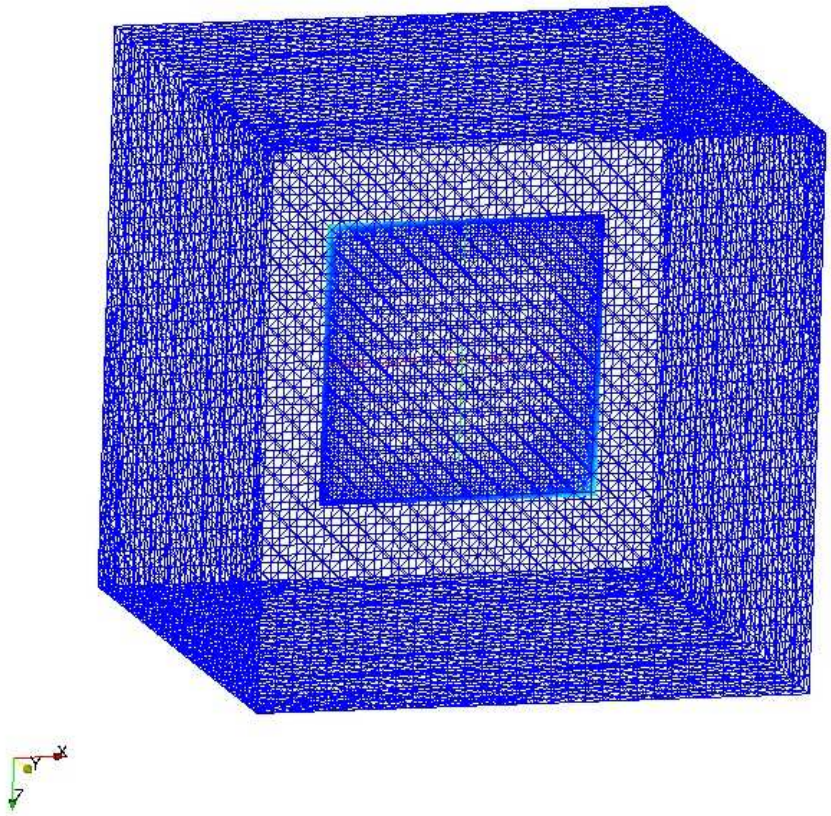

图 4.9 网格透视图（算例二）

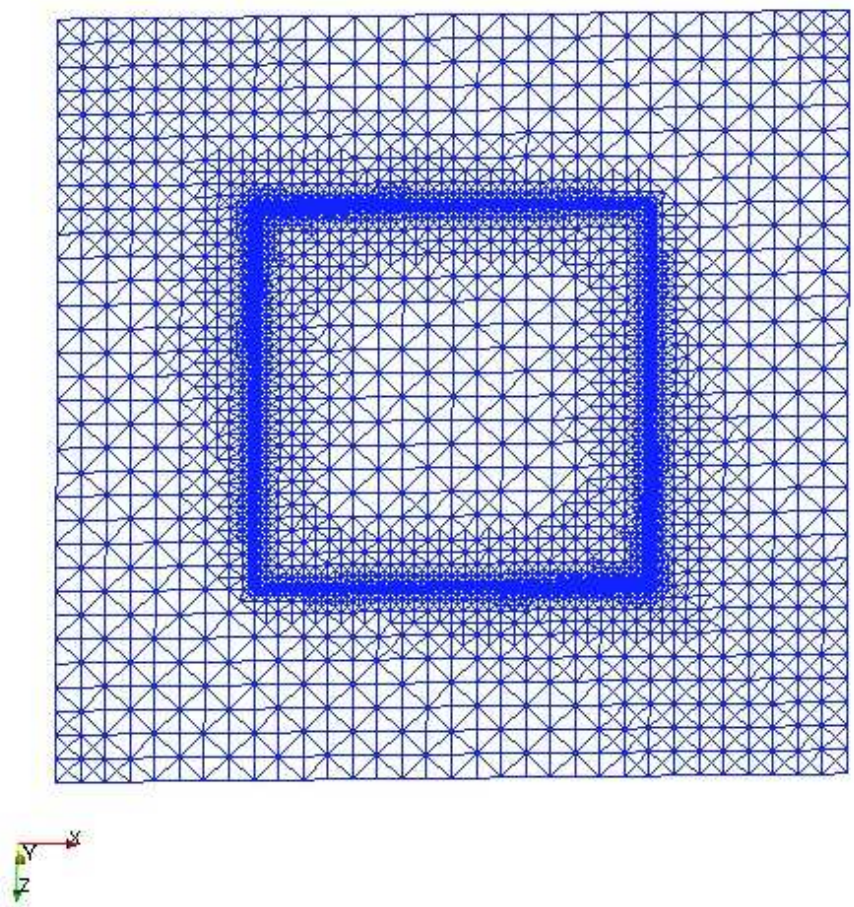

图 4.10 $\sum$ 平面的网格图（算例二）

表 4.7 自适应细化的情况，迭代终止残量为 1.0E-10（算例二，128 结点）

网格层号	自由度数/单元数	后验误差估计值	求解时间（秒）/迭代步数（次）	单结点内存使用量 (MB)
1	612/384*(3)	9.329e-01	0.0427/236	32.78
2	2078/1568*(15)	5.878e-01	0.4664/1012	32.76
3	4082/3044*(30)	4.517e-01	0.7859/1166	32.75
4	11124/8732*(87)	3.320e-01	3.1713/2666	33.01
5	37098/29832*(128)	2.179e-01	11.6157/9715	33.22
7	123226/100084	1.635e-01	46.8608/46256	33.36
11	2475940/2073216	5.768e-02	15449.6995/1530669	46.76
12	3269728/2735852	5.339e-02	52158.7333/4000000	49.75
13	6179278/5155072	4.585e-02	96981.5397/4000000	64.96
14	16229360/13668260	3.245e-02	259210.0547/4000000	125.5

* () 内注明实际使用的处理器数

第 11,12,13 个自适应步求解线性系统时迭代次数均达到了最大迭代步数，迭代退出时误差残量为 1.05e-09、8.603e-10 和 1.555e-09。

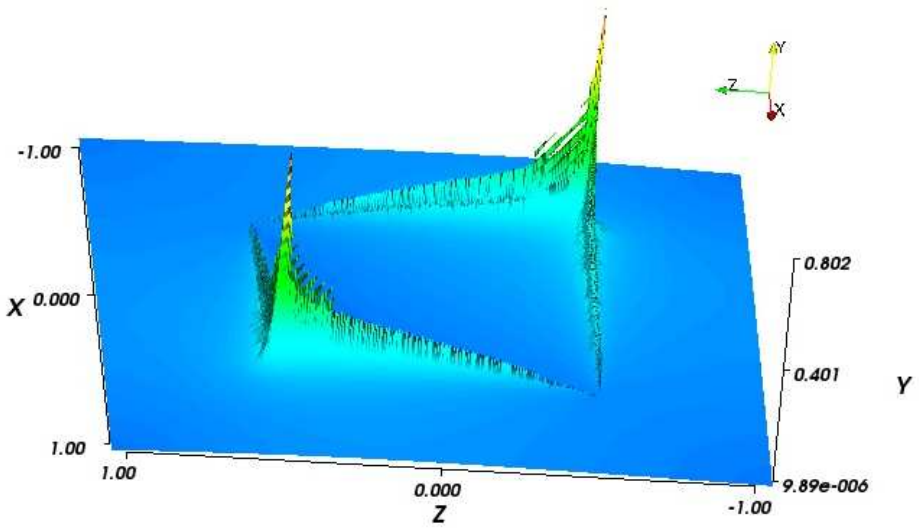

图 4.11 $\sum$ 平面上解向量的模（算例二）

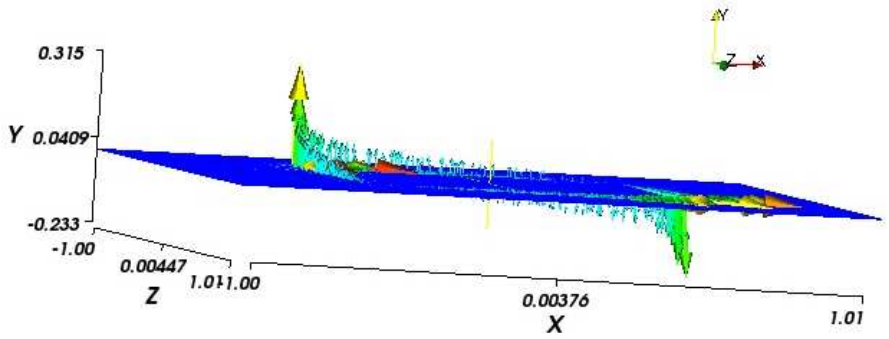

图 4.12 $\sum$ 平面上解向量（算例二）

§4.4.3 算例三

取 $\Omega_1=(-1,1)^3$，$\Omega_2=(0,1)\times(0,-1)\times[-1,1]$，$\Omega=\Omega_1\backslash\overline{\Omega}_2$，$\mu=1$, $k^2=1$。在 Ω 上取真解为

$$\vec{u}=\nabla(r^{\frac{1}{2}}\sin(\frac{\phi}{2}))$$

其中 (r,ϕ) 为柱面坐标。边界上取 Dirichlet 边界条件 $\vec{u}\times\vec{n}=g$。$\vec{u}$ 在 L 型区域的 z 轴上具有很强的奇性。

表 4.8 一致细化的情况，迭代终止残量为 1.0E-10（算例三，64 结点）

网格层号	自由度数/单元数	$\|u-u_h\|_{H(\mathrm{curl},\Omega)}$	后验误差估计值	求解时间（秒）/迭代步数（次）
1	81/36*（1）	8.847e-01	4.753e-01	0.0050/16
2	480/288*（2）	6.547e-01	3.499e-01	0.0317/292
3	3240/2304*（23）	4.689e-01	2.529e-01	0.4373/894
4	23664/18432*（64）	3.337e-01	1.807e-01	2.8175/3899
5	180576/147456	2.367e-01	1.284e-01	11.3041/6988
6	1410240/1179648	1.676e-01	9.102e-02	206.1461/17788
7	11145600/9437184	1.186e-01	6.444e-02	4132.2952/43772

* () 内注明实际使用的处理器数

表 4.9 自适应细化的情况，迭代终止残量为 1.0E-10（算例三，64 结点）

网格层号	自由度数/单元数	$\|u-u_h\|_{H(\mathrm{curl},\Omega)}$	后验误差估计值	求解时间（秒）/迭代步数（次）
1	81/36*（1）	8.847e-01	4.753e-01	0.0274/16
2	408/250*（2）	6.529e-01	3.512e-01	0.0638/296
3	1447/1066*（10）	4.782e-01	2.651e-01	0.4316/1106
4	3760/2916*（29）	3.538e-01	2.023e-01	1.7845/2506
5	8625/6856*（64）	2.668e-01	1.572e-01	4.9994/5246
6	19644/15776	2.044e-01	1.248e-01	7.9875/9236
9	261868/215568	8.716e-02	5.652e-02	344.8600/154168
12	3310072/2759768	3.777e-02	2.533e-02	30140.0202/1170362

* () 内注明实际使用的处理器数

表 4.8 和表 4.9 分别给出了使用一致网格细化和自适应网格细化时，在不同网格层的有限元插值误差 $\|u-u_h\|_{H(\,\mathrm{curl},\Omega)}$ 和后验误差估计值以及求解时间。从中可以看到，一致网格细化在单元数为 147456 时有限元插值误差 $\|u-u_h\|_{H(\,\mathrm{curl},\Omega)}$ 为 2.367e-01，而自适应网格只需要 15776 个单元便能使有限元插值误差 $\|u-u_h\|_{H(\,\mathrm{curl},\Omega)}$ 达到 2.044e-01。当一致网格细化中单元数达到 9437184 个时，有限元插值误差和自适应后验误差估计的值分别为：1.186e-01 和 6.444e-02，而在自适应网格中，单元数为 2759768 个时，有限元插值误差和自适应后验误差估计值分别为 3.777e-02 和 2.533e-02。从图 4.15 和图 4.16 可以清楚地看到，自适应网格很好地抓住了问题的奇性，同时图 4.13 和图 4.14 显示，自适应细化的网格中后验误差估计和有限元插值误差基本达到了最优的收敛阶，而全局细化做不到这一点。

图 4.17 和图 4.18 给出了在 $x=0$ 平面上的真解向量的模和有限元解向量的模。图 4.19 和图 4.20 分别给出了在 $x=0$ 平面上的真解向量和有限元解向量。

这些说明我们程序的棱单元模块和自适应模块是非常有效的。而且用户程序的代码只有 200 行左右，棱单元模块以及并行相关的内容都被很好地封装在了 PHG 库中，使得用户程序与串行程序一样简洁。

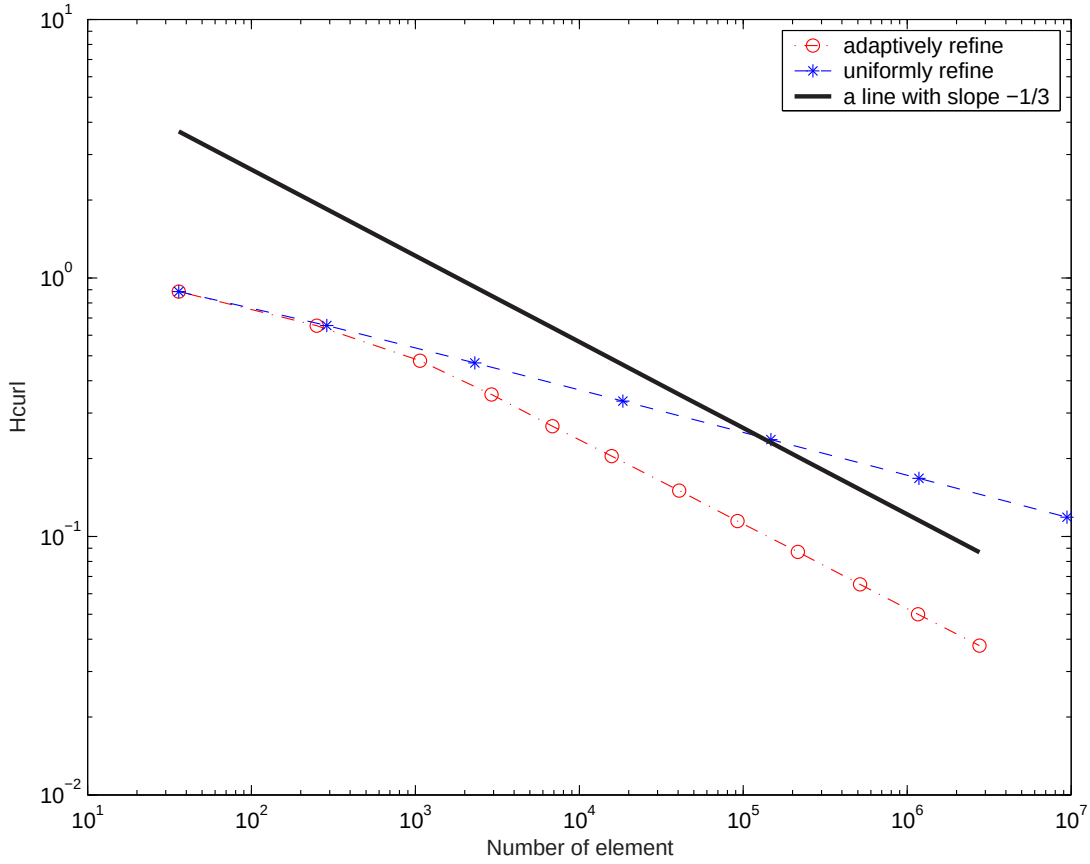

图 4.13 $\|u-u_h\|_{H(\mathrm{curl})}$ 在一致网格细化和自适应网格中的减小（算例三，LSSC-II 64 结点）

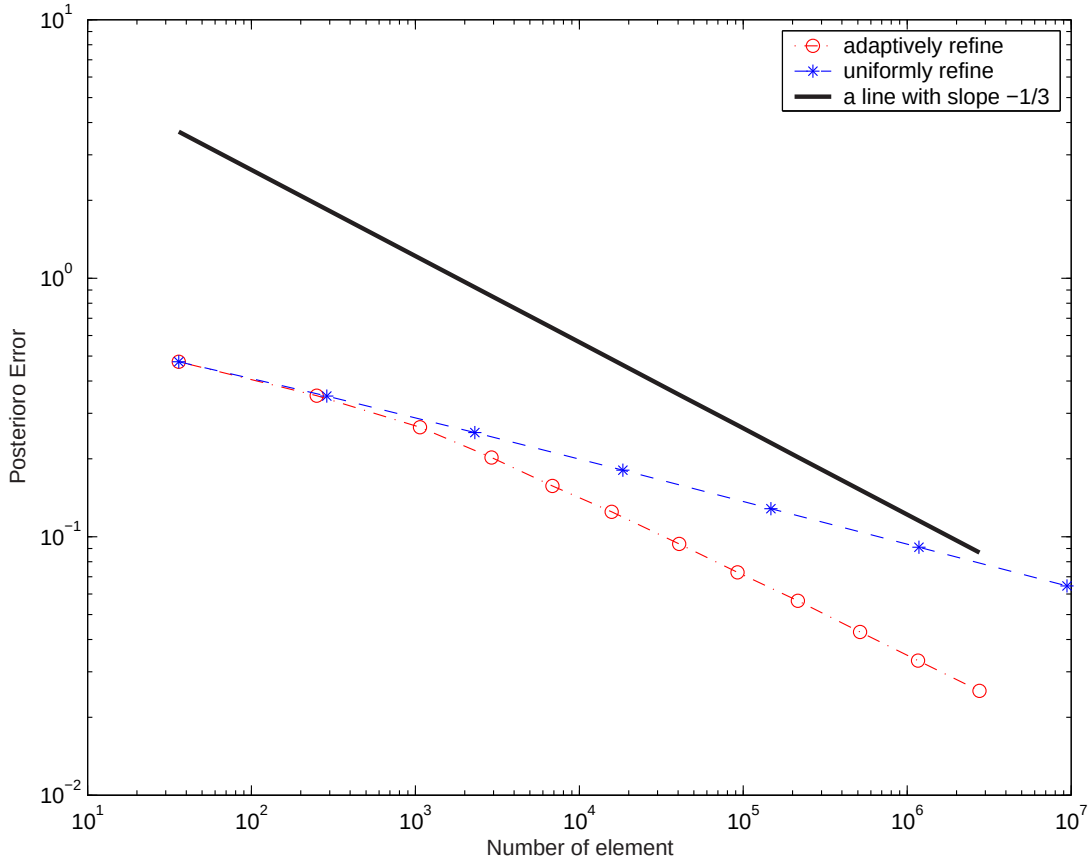

图 4.14 后验误差估计子在一致网格细化和自适应网格中的减小（算例三，LSSC-II 64 结点）

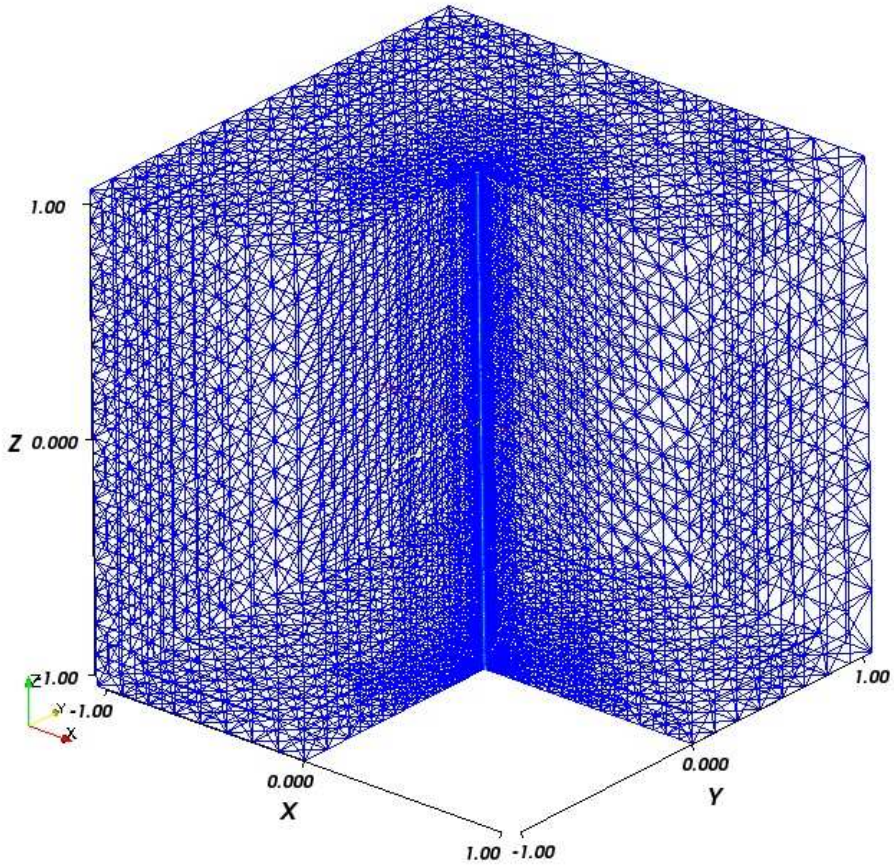

图 4.15 网格全图（算例三）

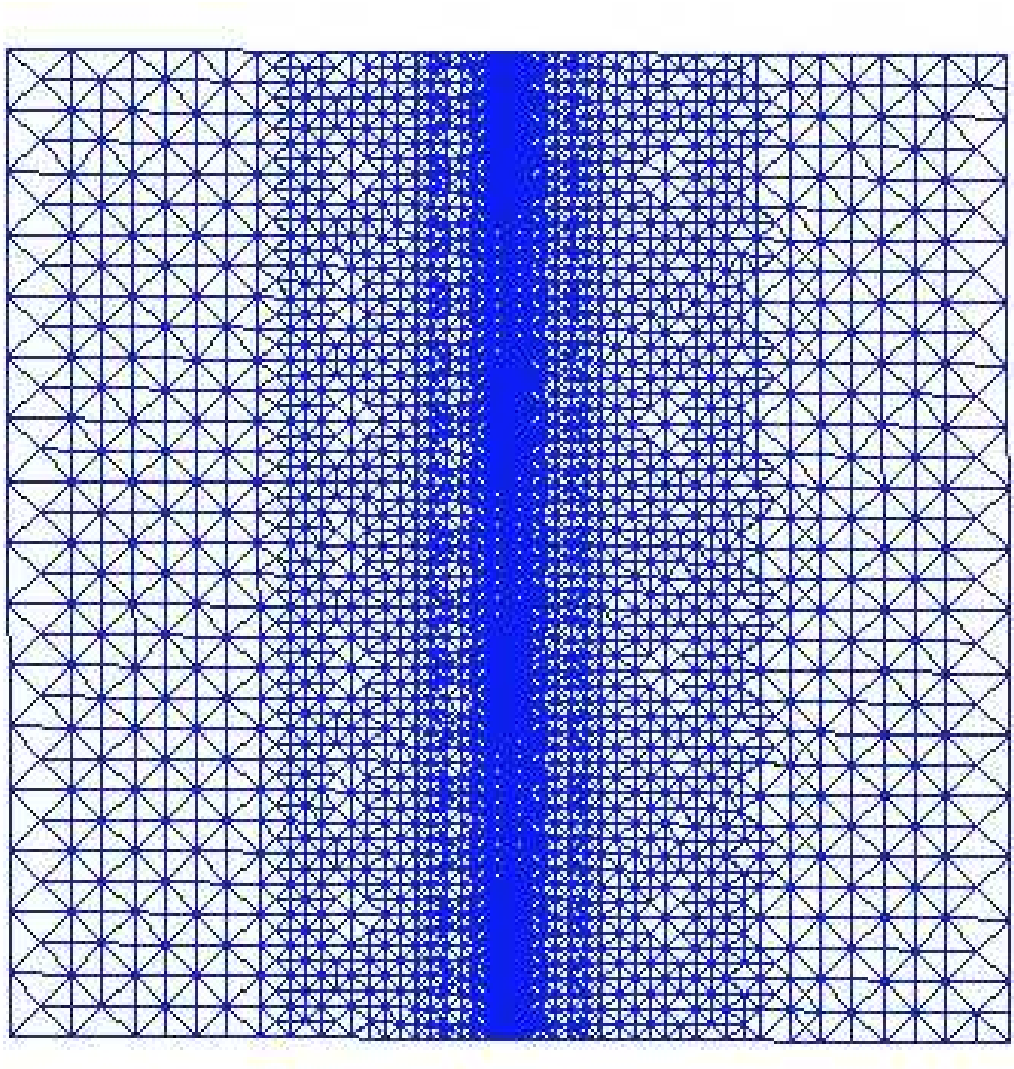

图 4.16 $x = 0$ 平面的网格图（算例三）

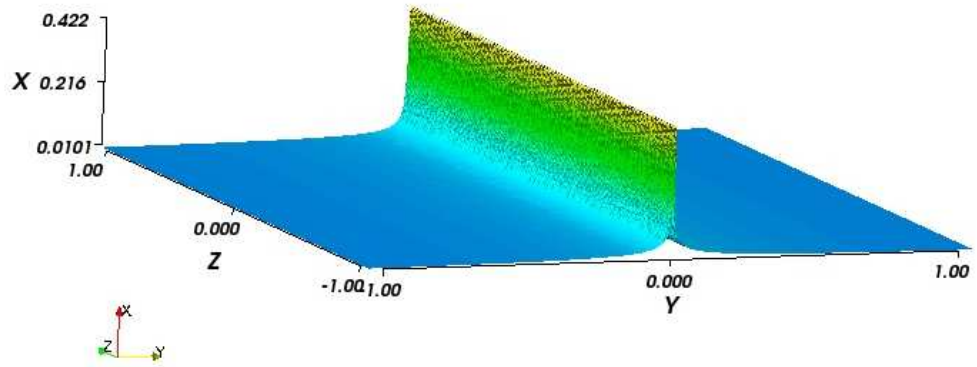

图 4.17 $x = 0$ 平面上真解向量的模（算例三）

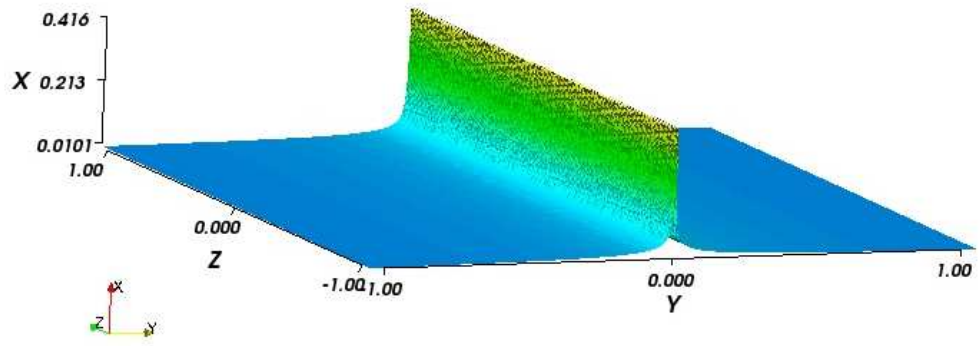

图 4.18 $x = 0$ 平面上有限元解向量的模（算例三）

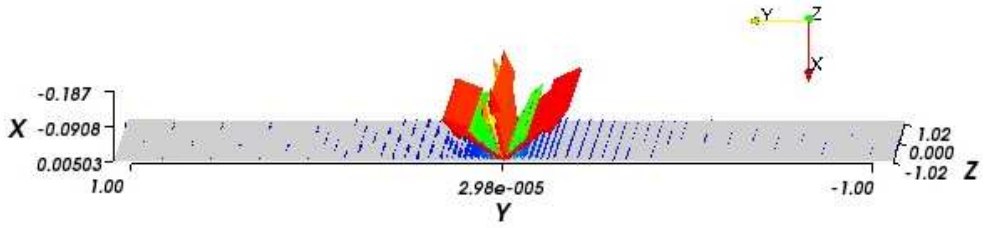

图 4.19 $x = 0$ 平面上真解向量（算例三）

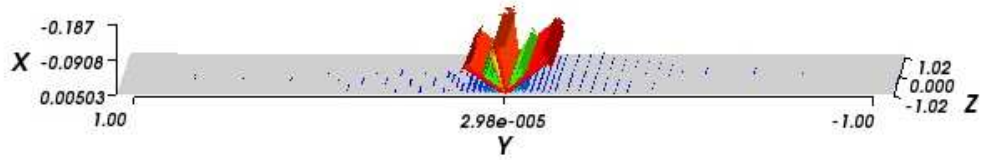

图 4.20 $x = 0$ 平面上有限元解向量（算例三）

第五章　结论及展望

本文以尚处在初始开发阶段的三维自适应有限元软件平台 PHG 为基础，实现了线性棱单元，并编写了三维时谐 Maxwell 方程的并行自适应有限元计算程序，完成了三个算例的并行数值模拟。程序采用标准 C 语言编制。在这个过程中，我们修正了 PHG 中原有代码在处理矢量基函数时的一些问题，并设计和修改了一些相关的数据结构和函数接口，改进了 PHG 的数值积分函数的基函数值 cache 机制使其接口更加规范，适用性更强。通过大量的数值实验验证了程序的正确性，考查了程序在大规模并行中的并行效率及并行可扩展性。线性棱单元在 PHG 中的成功实现为开展电磁场问题的大型数值模拟以及将来高阶棱单元的实现打下了很好的基础。

值得一提的是，在 PHG 基础上编写的时谐场并行自适应有限元的用户程序只需要大约 200 行代码，并且完全不用关心与并行有关的内容，代码甚至比一些串行有限元程序还要简单。

从数值实验中，我们发现自适应网格生成的线性系统性态很差，求解时使用简单的预条件子不能保证 LGMRES 方法快速收敛。当问题规模达到千万个单元以上时，即使做并行计算，其求解时间也远远太长。因此在未来工作中，在 PHG 中实现多重网格预条件子以及研究其它针对 Maxwell 方程所形成的线性方程组的有效算法是非常必要的。

参考文献

[1] 冯康，基于变分原理的差分格式，应用数学与计算数学，第 2 卷，第 4 期，238（1965）

[2] Babuška I. and Rheinboldt W., Error Estimates for Adaptive Finite Element Computation, SIAM J. Number. Anal., 15(1978), 736–754

[3] A. H. Baker, E. R. Jessup, T. Manteuffel, A Technique for Accelerating the Convergence of Restarted GMRES, SIAM J. Matrix Analysis and Applications, 26(2005), 962–984

[4] Rüdiger Verfürth, A Review of A posteriori Error Estimation and Adaptive Mesh-Refinement Techniques, Viley, Teubner

[5] R. Beck, R. Hiptair, R. H. W. Hoppe and B. Wohlmuth, Resiual Based A posteriori Error Estimators for Eddy Current Computation, ESAIM-Math. Model. Numer. Anal., vol 34, 159–182, Jan/Feb 2000

[6] R.Beck, R. Hiptair, R. H. W. Hoppe and B. Wohlmuth, Adaptive Multilevel Methods for Edge Element Discretizations of Maxwell's Equations, Surv. Math. Ind., 8(1999), 271–312

[7] O. Boffi, P. Fernandes, L. Gastaldi, I. Perugia, Computational Models of Electrmagnetic Resonators: Analysis of Edge Element approximation, SIAM J. Numer. Anal., 36(1990), 1264–1290

[8] J. Jackson, Classical Electrodynamics, 2nd ed., John Wiley, New York, 1975

[9] MPI — Message Passing Interface, `http://www-unix.mcs.anl.gov/mpi/`

[10] MPICH — An Implementation of MPI, `http://www-unix.mcs.anl.gov/mpi/mpich/`

[11] LAM-MPI — An Implementation of MPI, `http://www.lam-mpi.org/`

[12] Rajat Aggarwal, Kirk Schloegel, Vipin Kumar and Shashi Shekhar, METIS — Family of Multilevel Partioning Algorithms, `http://www-users.cs.umn.edu/~karypis/metis/`

[13] Rajat Aggarwal, Kirk Schloegel, Vipin Kumar and Shashi Shekhar, PARMETIS — Parallel Graph Partitioning and Fill-reducing Matrix Ordering, `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/publications`

[14] Satish Balay, William D. Gropp, Lois Curfman McInnes and Barry F. Smith, PETSc — Portable，Extensible Toolkit for Scientific Computation, `http://www-unix.mcs.anl.gov/petsc/petsc-2/`

[15] S. Blazy, J. Hungersher, and S. Schamberger, PadFEM —- Parallel Adaptive FEM, http://www.uni-paderborn.de/cs/padfem/

[16] Alfred Schmidt and Kunibert G. Siebert, ALBERT — An Adaptive Hierarchical Finite Element Toolbox, http://www.mathematik.uni-freiburg.de/IAM/Research/projects/albert/

[17] Manish Parashar, James C. Browne, DAGH: Data-Management for Parallel Adaptive Mesh-Refinement Techniques, http://www.caip.rutgers.edu/~parashar/DAGH/

[18] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, C. Wieners: UG — A Flexible Software Toolbox for Solving Partial Differential Equations, Computing and Visualization in Science 1, 1997, 27–40

[19] Paraview — Parallel Visualization Application, http://www.paraview.org/

[20] MayaVi, http://mayavi.sourceforge.net

[21] W. Dörfler, A Convergent Adaptive Algorithm for Possion's Equations, SIAM J. Numer. Anal., 33(1996), 1106–1124

[22] J, Jackson, Classical Electrodynamics, 2nd ed., John Wiley, New Yorkxi, 1975

[23] P. P. Silvester, R. L. Ferrari, Finite Element for Electrical Engineers, Cambridge University Press, Cambridge, 1990

[24] B. Dillon, J.Webb, A Comparison of Formulations for the Vector Finite Element Analysis of Waveguide, IEEE Trans. McGill University, Montreal, 1988

[25] Zhang Lin-bo, A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection, Preprint ICM–05–09, March 22, 2005

[26] A. Liu, B. Joe, Quality Local Refinement of Tetrahedral Meshes Based on Bisection, SIAM J. Sci. Comput., 16(1995), No. 6, 1269–1291

[27] D. N. Arnold, A. Mukherjee, and L. Pouly, Locally Adapted Tetrahedral Meshes Using Bisection, SIAM J. Sci. Comput., 22(2000), No. 2, 431–448

[28] I. Kossaczky，A Recursive Approach to Local Mesh Refnement in Two and Three dimensions, J. Comput. Appl. Math., 55(1994), 275–288

[29] H. Whitney, Geometric Integration Theory, Princeton, NJ:Princeton University Press, 1957

[30] J. C. Nédélec, Mixed Finite Elements in R^3, Numer. Math., 35(1980), 315–341

[31] J. C. Nédélec, A New Family of Mixed Finite Elements in R^3, Numer. Math., 50(1986), 57–81

[32] A. Bossavit, J. C. Verite, A Mixed FEM-BIEM Method to Solve 3-D Eddy Current Problems, IEEE Trans. Magn., vol. MAG-18, 1982, 431–435

[33] M. L. Barton, Z. J. Cendes, New Vector Finite Elements for Three-dimensional Magnetics Field Computation, J. Appl. Phys., 61(1987), No. 8, 3919–3921

[34] Gauss, C. F., Methodus nova integralium valores per approximationem inveniendi. Commentationes Societatis regiae scientarium Gottingensis recentiores 3, 39-76, 1814. Reprinted in Werke, Vol. 3. New York: George Olms, p. 163, 1981.

[35] Gaussian Quadrature,
`http://mathworld.wolfram.com/Legendre-GaussQuadrature.html`

[36] Radau Quadrature
`http://mathworld.wolfram.com/RadauGaussQuadrature.html`

[37] D. Dunavant, High Degree Efficient Symmetrical Gaussian Quadrature Rules for the Triangle, Int. J. Numer. Method Eng., 21(195), 1129–114

[38] H. Kardestuncer, ed., Finite Element Handbook, McGraw-Hill, New York, 1987

[39] A. H. Stroud, Approximate Calculation of Multiple Integrals, Prentice-Hall, Englewood Cliffs, NJ, 1971

[40] M. V. Wilkes, Slave Memories and Dynamic Storage Allocation, Trans. IEEE, Vol. EC-14, No. 2, Apr 1965, 270–271

[41] R. F. Harrington, Time-Harmonic Electromagnetic Fields, New York, McGraw-Hill, 1961

[42] J. Jin, The Finite Element Method in Electrodynamics, John Wiley, New York, 1993

[43] P. Monk, A Finite Element Method for Approximating the Time-harmonic Maxwell Equations, Numer. Math., 63(1992), 243–261

[44] P. Monk, Analysis of a Finite Element Method for Maxwell's Equations, SIAM J. Numer. Anal., 29(1992), 714–729

[45] 王龙，Maxwell 方程组的自适应多重网格计算，中国科学院研究生院博士学位论文

[46] 金建铭，电磁场有限元方法，西安电子科技大学出版社，1998

[47] Jack Dongarra 著，莫则尧，陈军，曹小林译，并行计算综论 (Sourcebook of Parallel Computing), 电子工业出版社，2005

[48] 莫则尧，袁国兴，消息传递并行编程环境 MPI，科学出版社，2001

[49] 都志辉，高性能计算并行编程技术 MPI 并行程序设计，清华大学出版社，2001

[50] 谷同祥，大型稀疏线性代数方程组的并行非定常迭代方法，博士论文

[51] 谷同祥，并行数值分析软件综述，博士后研究工作报告

[52] 曹志浩，数值线性代数，复旦大学出版社，1996

[53] 胡健伟，汤怀民，微分方程数值解法，科学出版社，1999

[54] G. H. 戈卢步，C. F. 范洛恩著，袁亚湘译，矩阵计算，1999

[55] 周树荃，梁维泰，邓绍忠，有限元结构分析并行计算，科学出版社，1999

[56] 张宝淋，谷同祥，莫则尧，数值并行计算原理与方法，国防工业出版社，1999

致　谢

值此论文完成之际，我要衷心的感谢我的导师张林波研究员。回首过去的三年时光，是他将我引入了一个充满趣味而又富有挑战的研究方向，是他给予我指导、关怀和支持。我的每一点进步，都凝聚着张老师的心血。他的治学态度严谨，对科研工作精益求精；他知识广博，思维敏锐，与他讨论使我终生受益；他忘我工作，兢兢业业的精神催人奋进。能成为他的学生，是我今生一大幸事，在此向张老师致以诚挚的谢意!

感谢陈志明老师一直以来对于我的关心和鼓励，他的治学态度和处世风格对我影响极深。

感谢袁亚湘老师、周爱辉老师、曹礼群老师、洪佳林老师、李忠泽老师和郑伟英老师在学习和生活上给予帮助与支持!

感谢计算数学与科学工程计算研究所提供的舒适的学习环境和良好的学术氛围。本文的工作是在科学与工程计算国家重点实验室完成的，感谢实验室提供的优越的科研环境和高水平的计算环境。感谢白英老师，感谢吴继萍老师、樊建荣老师、张纪平老师和丁如娟老师的关心和帮助。感谢数学与系统科学研究院人教处邵欣老师、尹永华老师和关华老师。

感谢我的师兄弟们。郭小虎、刘青凯和刘辉，感谢它们在许多问题上给予我的宝贵经验和具体的帮助。刘青凯师兄和刘辉师弟对我的论文提出了很多有建设性的意见。感谢王龙师兄、陈俊清师兄在学业上给予我的帮助，不厌其烦的解答我的问题，对于我的论文提出很多宝贵的意见。

在三年的学习和生活里，结识了很多真诚的朋友，王辛、戴小英、徐姿、吴新民、马士谦、谢和虎、张海樟、郭绍俊、夏勇、殷俊峰、刘歆等不一一而数了，感谢你们给予我的无私帮助。

最后谨以此文献给我的父母，祝愿他们身体健康!

索　引